Replication in Data Stream Management Systems

ChronicleDB on a Raft

Christian Konrad

August 15th, 2022

Declaration

I hereby declare that the work is entirely my own and was produced with no assistance from third parties.
I certify that the work has not been submitted in the same or any similar form for assessment to any other examining body and all references, direct and indirect, are indicated as such and have been cited accordingly.

Erklärung

Ich versichere hiermit, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde.
Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Christian Konrad
Marburg, August 15th, 2022

Acknowledgements

This work would not have been possible without the patience of many people around me, who have backed off while I was busy working on this thesis day and night. Things were a bit complicated when I was writing this thesis and doing the preliminary work, so I am all the more grateful for their support. This includes my family, my partner, my son, and my friends. Now that this is done, I turn my attention back to you (at least until the next big thing catches my attention).

I thank my reviewers Nikolaus Glombiewski and Michael Körber for their continuous support and for never getting frustrated when I did not manage to respond in a timely manner. Their advice about not letting the scope of this work get too large was important for me to stay focused (but I still have a lot to talk about, as working on this topic really put me in a flow at times).

Many thanks to Alexander Markowetz for advising me not only regarding this thesis, but also on general career questions that helped me figure out what I want to be about in the next few years and how to settle into a field that will satisfy me in my future career. He is a great person to talk to—and a pleasure to listen to.

Thanks to my partner Sarah for motivating me to finish my master's degree and write this thesis. She was very insistent—and I'm grateful to her for that, because otherwise you wouldn't be reading this now.

Abstract

ChronicleDB is a high-performance event store designed for the age of IoT and data-intensive event stream processing. It runs embedded in other applications on low-cost hardware, such as sensors, and standalone on individual machines, significantly outperforming the competition for both writes and queries. Unfortunately, ChronicleDB cannot be deployed on a distributed cluster with high availability, scalability and fault-tolerance, since it does not provide a data replication layer.

To bridge the gap and enable ChronicleDB to run in full edge-to-cloud networks, we present an approach for a distributed ChronicleDB in this work. To accomplish this, we collected requirements for the various use cases of distributed data stream management systems. We investigated dependability, fault-tolerance, and consistency models under the conditions of the CAP and PACELC theorems, and compared state-of-the-art implementations in the industry. We finally chose the Raft consensus algorithm as it satisfies these requirements. We implemented a prototypical replication and partitioning layer for ChronicleDB based on our decisions, and evaluated it in terms of the trade-offs to be made.

Our evaluation has shown that we can achieve fault-tolerance, availability, and scalability, but at the cost of lower throughput for a single event stream. We also pointed out potential optimizations that can reduce replication costs and overall write and read latency, which can be helpful in developing a production-ready distributed ChronicleDB.

Zusammenfassung

ChronicleDB ist ein hochleistungsfähiger Ereignisspeicher, der für das Zeitalter von IoT und datenintensiver Ereignisstromverarbeitung entwickelt wurde. ChronicleDB läuft eingebettet in anderen Anwendungen auf kostengünstiger Hardware, wie z. B. Sensoren, und eigenständig auf einzelnen Maschinen und übertrifft die Leistung der Konkurrenz sowohl bei Schreibvorgängen als auch bei Abfragen erheblich. Leider kann ChronicleDB nicht in einem verteilten Cluster mit hoher Verfügbarkeit, Skalierbarkeit und Fehlertoleranz eingesetzt werden, da es keine Datenreplikationsschicht bietet.

Um diese Lücke zu schließen und ChronicleDB den Betrieb in vollständigen Edge-to-Cloud-Netzwerken zu ermöglichen, stellen wir in dieser Arbeit einen Ansatz für einen verteilten ChronicleDB-Ereignisspeicher vor. Um dies zu erreichen, haben wir die Anforderungen für die verschiedenen Anwendungsfälle von verteilten Datenstrommanagementsystemen gesammelt. Wir untersuchten die Begriffe Zuverlässigkeit, Fehlertoleranz und Konsistenzmodelle unter den Bedingungen der CAP- und PACELC-Theoreme und verglichen die neuesten Implementierungen von verteilten Ereignisspeichern, Zeitreihendatenbanken und vergleichbaren Systemen. Wir entschieden uns schließlich für den Consensus-Algorithmus Raft, da er diese Anforderungen erfüllt. Auf der Grundlage unserer Entscheidungen haben wir eine prototypische Replikations- und Partitionierungsschicht für ChronicleDB implementiert und sie im Hinblick auf die zu treffenden Kompromisse bewertet.

Unsere Evaluierung hat gezeigt, dass wir Fehlertoleranz, Verfügbarkeit und Skalierbarkeit erreichen können, allerdings auf Kosten eines geringeren Durchsatzes für einen einzelnen Ereignisstrom. Wir haben auch potenzielle Optimierungen aufgezeigt, die die Replikationskosten und die gesamte Schreib- und Leselatenz verringern können, was bei der Entwicklung einer produktionsreifen verteilten ChronicleDB hilfreich sein kann.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Distributed event-based systems are the key to increase the scalability of today's information systems [1]. At the foundation of data-intense applications are distributed event stores that reside in cloud architectures, but also stretch beyond devices in the cloud and locally to form *edge-cloud networks* [2]. Both research and enterprise applications today require systems that provide low latency, high availability and real-time monitoring capabilities, many of them powered by the *Internet of Things* (IoT) and *Complex Event Processing* (CEP [1]), such as industrial big data and smart factories [3], [4], high-bandwidth clinical and health applications including sensor data from smart wearables [5], self-driving cars, smart cities, smart energy grids [6], or environments supporting scientific experiments [7], [8] (e.g. at the CERN large hadron collider) to name a few. Typically, those systems consume a huge amount of events generated by millions of users, sensors and devices. In addition, the rise of 5G networks drives innovation in this area and therefore increases the need for those systems to seize this opportunities in full extend [9].

1.1 Motivation

Due to these increasing demands, such event-based systems must therefore be able to write events arriving at very high and fluctuating rates to persistent storage and support ad hoc analytical queries. Since conventional database systems are not able to provide the required write performance, novel write-optimized data stores such as log-based systems or key-value stores have been introduced recently. However, the drawbacks of these systems are mediocre query performance and the lack of appropriate mechanisms for immediate recovery in case of system failures. As a result, event stores emerged, such as ChronicleDB, a novel high-throughput

database system with a storage layout tailored to provide high write performance for fluctuating data volumes and powerful indexing capabilities to support a wide range of queries. ChronicleDB has been shown to outperform competing systems in both write and read performance [10]. At present, ChronicleDB is designed either as an embeddable library for tight integration into an application or as a standalone database server, but without the ability to run in a scalable cluster architecture.

While event stores running locally or embedded on IoT devices must be ultra-lightweight and fast, when deployed to the cloud, they must provide strong dependability characteristics. Furthermore, a smooth interplay between this different types of deployments is crucial: the cloud-deployed store must be able to reliably ingest a huge amount of different streams, with both different or equal event schemas, that are fed into it by the embedded systems.

With edge computing, the requirements on today's distributed systems increase significantly: they must be able to maintain high throughput, store a huge amount of data efficiently, and respond almost immediately to user queries, while providing the desired level of fault-tolerance, consistency, and high availability. All of these requirements must be met while serving millions of users at the same time.

Approaching a solution to this problems leads to *data replication*. Replication of data resources both within a local computing cluster and across different geographic regions ensures availability and consistency to a certain degree in the event of a node failure resulting from various types of faults. An effective replication protocol not only enables efficient access to vast amounts of data, but also reduces latency in large-scale, geo-replicated cloud environments. However, there is no one-size-fits-all solution: applications that rely on replication for high availability and low latency must make trade-offs between consistency, availability, and latency.

Fortunately, several replication algorithms have already been proposed to achieve availability under different consistency and fault-tolerance conditions, such as the *Raft consensus protocol*. This work attempts to identify such a replication protocol that meets the desired write and query performance characteristics for ChronicleDB, and then find an implementation design so that it can be used as a fault-tolerant, highly available, and scalable cluster in addition to the embedded and standalone server modes.

1.2 Problem Formulation

This brings us to the subject of this thesis. The objective is to move an existing standalone, non-distributed event repository into a distributed system with fault

tolerance and high availability. To achieve this goal, we attempt to find and evaluate a replication algorithm that meets the desired performance and dependability requirements. To find such an algorithm, we use an exploratory research approach that includes both quantitative and qualitative methods to discover the possible solution space:

- Identification of consistency models suitable for an event store with a specific set of dependability requirements and differentiation from other consistency models including a detailed discussion of the advantages and disadvantages of the models in question.
- Identification and justification of a replication protocol suitable for an event store that satisfies the characteristic requirements of the selected consistency models, including a differentiation from other replication techniques.
- Investigation and quantitative analysis of the performance trade-off of an implementation of the chosen replication protocol in an event store to meet the given dependability requirements, including fault-tolerance and horizontal scalability, compared with the expected trade-off, as well as an examination of the dependability properties themselves.

Additionally, the results of this work serve as a plausible grounded theory about providing such a replication layer, to build new hypotheses and opportunities on. Therefore, we answer the following research questions:

- What are the unique characteristics of event stores and how do they influence the needs and requirements for a replication layer to make them fault-tolerant?
- What are the advantages and disadvantages of the different replication protocols for the ChronicleDB event store?
- What is the performance and throughput impact and what influences it? What are ways to improve it?
- Why is there a special interest in this research?

Positive findings from this research work would provide worthwhile benefits to distributed event stores:

- The impact of the application of different consistency models and replication protocols to event stores is known and described and helps both academics and developers to decide which protocols to go for, depending on their use cases and requirements.

- Users can operate event stores in an edge-cloud architecture, leveraging the different characteristics of embedded, single-node vs. replicated, multi-node clusters in different places of their system design, to be able to deal with the high throughputs occuring in heavily distributed IoT systems.
- Open questions and known problems are identified and documented properly so that the performance can be further improved to meet the requirements of such event store architectures used in production.

1.3 Contribution

To the best knowledge, this work is the first attempt published in academia focusing on applying the Raft consensus protocol to event stores. There are a few event stores and time series databases used in industry that leverage Raft to achieve fault-tolerance and scalability (InfluxDB, IoTDB), but replication and consensus were only mentioned as a side note in academic research on those systems.

In this thesis we discuss and analyse several different replication algorithms to find the one that fits our requirements for an event store. The contributions are:

- A thorough discussion of consistency models and replication protocols for a distributed event store ready for the edge-cloud and capable of processing very high data throughput. A state-of-the-art replication protocol is then selected to handle this requirements in a future-proof way.
- A systematic review of previous implementations of replication protocols, focusing on decisions related to consistency, dependability, levels of data replicated, and the replication protocols chosen.
- An implementation of a replicated ChronicleDB event store based on Apache Ratis [11], to serve as a learning base for evaluating the consistency model and replication protocol that we found most useful[1].

- Benchmark-based performance evaluations of the implementation on event-store-specific metrics (event throughput, query speed) to study the throughput and scalability of network architectures with different numbers of nodes.

[1]The code is available as open-source in the public domain, at https://doi.org/10.5281/zenodo.6991168.

1.4 Limitations

Due to the complexity of the distributed systems issue, we limit ourselves to finding and implementing a replication protocol that provides fault tolerance and a selected set of dependability requirements in a single data center, while extending the discussion to other areas and considerations for future work on which we can build. We limit our solution to specific use cases, recognizing that it will not be the ideal solution, and we even postulate that there is no ideal solution.

1.5 Outline

The remainder of this work is structured as follows: Chapter 1 introduces the reader to the research context of this work, examines the Raft Consensus Protocol and the ChronicleDB event store, and discusses recent literature in this area. Chapter 2 describes the methodology underlying this research, presents the main implementation choices and compares recent work. Chapter 3 then illustrates the results of the evaluation of the implementation. Chapter 4 finally draws conclusions from this work and outlines recommendations, key learnings, weaknesses of this approach and future challenges.

Chapter 2

Fundamentals

This chapter introduces the reader into the fundamentals of replication and event stores. The chapter itself is divided into 4 sections: Section 1.1 explores the realm of distributed systems in general and replication in detail, comparing the various protocols, algorithms, and concepts to find a protocol that meets the requirements of event stores and in particular those of ChronicleDB. Event stores and time series databases are discussed and compared in (Section 1.3). Both sections are followed by a section dealing with either the protocol (Raft, Section 1.2) or event store (ChronicleDB, Section 1.4) of choice for this work in detail. The chapter concludes with an overview and provides the foundation for a deep understanding of the requirements, characteristics, trade-offs, and specifics of the protocols so that the reader can follow the decisions in the subsequent chapters to design and implement such a replicated system in an educated manner.

2.1 Replication

> A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.
>
> — Leslie Lamport

This section provides the reader with an overview of the topic of *distributed database systems* (DDBS), defines the term *replication*, explains why replication is crucial for many of those systems, discusses various replication protocols, and concludes with a literature review of recent research in this area. The goal of this section is to provide the reader with a thorough understanding of what replication

is, why, where, and when it is needed, and how replication protocols work.

A *distributed system* is a collection of autonomous computing elements that appears to its users as a single coherent system [12]. To achieve this behavior, such a system needs to offer certain levels of *availability*, *consistency*, *scalability* and *fault-tolerance*; therefore, many modern distributed systems rely heavily on replicated data stores that maintain multiple replicas of shared data [13]. Depending on the replication protocol, the data is accessible to clients at any of the replicas, and these replicas communicate changes to each other using message passing.

Thus, a distributed database system is a distributed system that provides read and write access to data. Replication in these databases takes place to varying degrees: from simple replication of metadata and current system state to full replication of all payload data, depending on the requirements and characteristics of the system.

Many modern applications, especially those served from the cloud, are inherently distributed. With edge computing, such applications can be distributed not only to nodes in one or more clusters serving many clients, but also to all of those clients participating in the edge-cloud network. Users of cloud applications expect the same experience independent of their geographic region and also regardless of the amount of data they produce and consume.

In complex distributed systems, there are multiple interoperating replicated *(micro)services* with different availability and consistency characteristics. Ensuring that these systems and services can reliably work together is the topic of container orchestration (e.g., through Kubernetes [14]) and service composition [15], [16]—which need to be replicated, too—which we will not discuss in this work.

Example use cases for replication in distributed systems (additionally to those mentioned in the Introduction) include large-scale software-as-a-service applications with data replicated across data centers in geographically distinct locations [17], applications for mobile clients that keep replicas near the user's location to support fast and efficient access [18], key-value stores that act as bookkeepers for shared cluster metadata [19], highly available domain name systems (DNS) [20], blockchains and all their various use cases (see Blockchain Consensus Protocols), or social networks where user content is to be distributed cost-effectively to millions of other users [21], to name a few.

The following subsections describe the fundamental concepts of *dependable* distributed systems, as well as use cases for replication and the challenges in these particular cases, before outlining relevant replication protocols.

2.1.1 Horizontal Scalability

Standalone applications can benefit from *scaling up* the hardware and leveraging concurrency techniques/multithreading behavior (in this case we speak of *vertical scalability*), but only to a certain physical hardware limit (such as CPU cores, network cards or shared memory). Approaching this limit, these applications and services can no longer be scaled economically or at all, as the hardware costs increase dramatically. In addition, applications running on only one single node[1] pose the risk of a single point of failure, which we want to counter with replication.

To scale beyond this limit, a system must be designed to be *horizontally scalable*, that is, to be distributable across multiple computing nodes/servers (also known as to *scale out*). Ideally, the amount of nodes scales with the amount of users, but to achieve this, certain decisions must be made regarding consistency models and partitioning. Legacy vertically scalable applications cannot be made horizontally scalable without rethinking the system design, which includes replication and message passing protocols between the nodes to which the application data is distributed.

With vertical scaling, when you host a data store on a single server and it becomes too large to be handled efficiently, you identify the hardware bottlenecks and upgrade. With horizontal scaling instead, a new node is added and the data is partitioned and split between the old and new nodes.

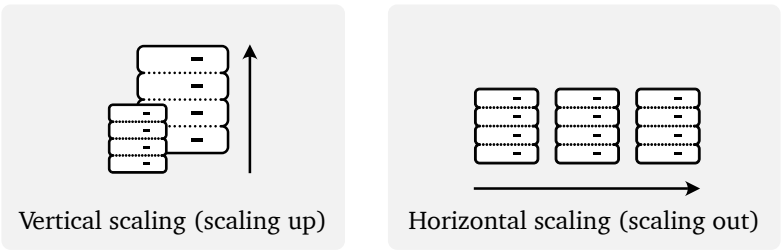


Figure 2.1: Vertical vs. horizontal scaling. a) With vertical scaling, existing machines are scaled up by upgrading. b) With horizontal scaling, more machines are added to handle and balance increased loads.

With an increasing number of users served, the number of nodes in a horizontally scalable system grows. In the ideal case, the number of nodes required to keep the perceived performance for users constant increases linearly with the number of

[1]Note that in this work, we use the terms *node* and *process* interchangeably when talking about replication.

distinct data entities served (such as tables or data streams with distinct schemas) [22]; in other words, the transaction rate grows linearly with the computational power of the cluster. This is not neccessarily the case when a increasing number of users access the same stream/table or inter-node/inter-partition transactions are executed. In the latter case, vertical scaling or advanced partitioning techniques [23] still come handy. In practise, many applications achieve near-linear scalability.

Opting in for horizontal scaling is beneficial for large-scale cloud and edge applications as it is easier to add new nodes to a system instead of scaling up existing ones, even if the initial costs are higher as the infrastructure and algorithms must be implemented in first place.

The system design that enables horizontal scalability is also a key requirement for replication. To achieve fault tolerance and availability through replication, it is required to store replicas of a dataset on multiple nodes, therefore the book keeping and message passing infrastructure that is neccessary for partitioning is also a basic requirement for replication protocols [13].

2.1.2 Safety and Reliability

> Anything that can go wrong will go wrong.
> — Murphy's Law

Modern real-time distributed systems are expected to be reliable, i.e., to function as expected without interruption, and to be safe, i.e., not to cause catastrophic accidents even if a subsystem misbehaves. For a distributed system to be both reliable and secure, it must be designed to be *fault-tolerant*, i.e., the application must be able to cope with node failures without interrupting service. In addition, it must also be able to withstand faults without operating incorrectly, i.e., responding to user requests with erroneous or malicious content. In large distributed systems, faults will happen - they are inevitable. Therefore, fault-tolerance is the realization and acknowledgement that there will be faults in a system. There is no system that is 100 % fault-tolerant.

Definition 2.1.1 (k-Fault-Tolerance)**.** A system that can tolerate at least k faulty nodes is called *k-fault-tolerant*.

To understand how a system can be designed to be both reliable and secure, we review the definitions of reliability and security [24]–[26]:

Definition 2.1.2 (Reliability)**.** The *reliability* of a system is the probability that its functions will execute successfully under a given set of environmental conditions

(operational profile[2]) and over a given time period t, denoted by $R(t)$. Here $R(t) = P(T > t), t \geq 0$ where T is a random variable denoting the time to failure. This function is also known as the *survival function*.

Different users using the same system in different ways may experience different levels of reliability. An operational profile shows how to increase productivity and reliability by allowing us to quickly find the faults that impact the system reliability mostly and by finding the right test coverage.

There are also other dimensions of interest to express the reliability of a system:

(1) The probability of failure-free operation over a specified time interval, as in above definition.

(2) The expected duration of failure-free operation.

(3) The expected number of failures per time interval (failure intensity).

The second can be measured using the *Mean Time Between Failures* (MTBF). The MTBF is a common and important measure and is especially critical for consensus protocols, a particular class of replication protocols, since the MTBF of a single node must be estimated in advance when defining the protocol's timing criteria, which will be shown later in Section 1.2 when we describe the replication protocol of choice of this work. The MTBF itself is a combined metric: it is the sum of the *Mean Time To Failure* (MTTF) and the *Mean Time To Repair* (MTTR).

Definition 2.1.3 (Mean Time To Failure)**.** The MTTF describes the mean time of uninterrupted *uptime* of a system where it is operational, therefore the average time between a succesful system start and a failure:

$$\text{MTTF} := \sum_i \frac{t_{\text{down},i} - t_{\text{up},i}}{n} = \sum_i \frac{u_i}{n} \tag{2.1}$$

where t_{down} is the start of the downtime after a failure, t_{up} the start of the uptime before that failure, u_i the current uptime period and n the number of failures.

This only describes the uptime: the *downtime* or repair time, including failure detection, reboot, error fixing and network reconfigurations are described by the MTTR. In some literature, the average amount of time it takes to detect a failure is

[2]An operational profile is a quantitative characterization of how a system will be operated by actual users

additionally extracted as a dedicated metric, the *Mean Time To Detect* (MTTD)[3].

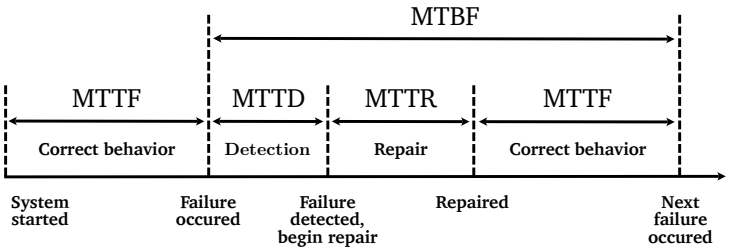


Figure 2.2: Relationship between the Mean Time Between Failure (MTBF) and other reliability metrics

Actually, the MTBF can also be expressed as an integral over the reliability function $R(t)$ [28], which illustrates the relation between the two metrics:

$$\text{MTBF} = \int_0^\infty R(t)dt$$

One of the most important design techniques to achieve reliability, both in hardware and software, is redundancy [29]. There are two main patterns to achieve redundancy: retry and replication. Retry is *redundancy in time*, while replication is *redundancy in space*. Using a retry strategy, a *failure detector* oftentimes is just a timeout. In this case, the MTTD is the timeout interval while the MTTR is the retry time. In replication, timeouts are also used to detect dead nodes (assuming a fail-stop failure model, as described in the following paragraphs) so they can be rebooted in the background or the network can be reconfigured (by the network itself or an external book keeper or orchestrator), commonly covered by the rules of the replication protocol and oftentimes without having a perceivable impact on reliability and availability for the user.

Definition 2.1.4 (Safety)**.** The safety of a system is the probability that no undesirable behavior will occur during system operation over a specified period of time. Safety looks at the consequences and possible accidents, and how to deal with it.

[3]Another metric, often used in manufacturing, is the *Mean Related Downtime for Preventive Maintenance* (MRDP), which describes a planned time of unavailability to conduct preventive actions that will actually reduce the net unavailability, as they reduce the risk of failures. In distributed systems, there are strategies for *hot updates* that allows to conduct preventive maintenance node by node without shutting down the whole system, as long as at least one replica set of nodes remains available, while other nodes are updating. This is limited to updates without breaking changes that disrupt interoperability of nodes. Modern messaging protocols like `Protocol Buffers` (which are used in this work) support this by providing schema-safety with both backward and forward compatibility [27].

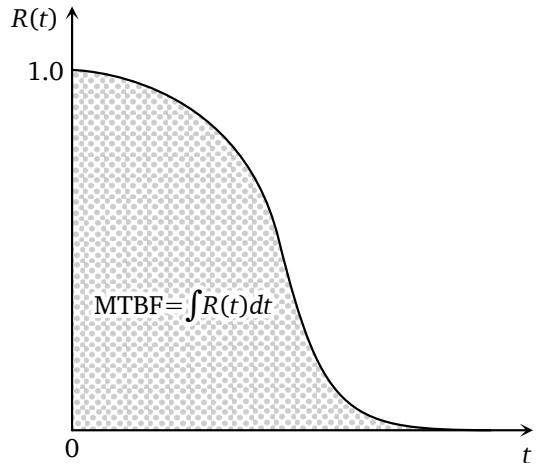


Figure 2.3: A curve for a typical R function, illustrating the relation between the R and MTBF metric

Safety is an important requirement especially for critical infrastructure applications. Safety means that the system behavior is as expected, and no faulty or even malicious results are returned.

A high degree of reliability, while necessary, is not sufficient to ensure safety. In fact, there can even be a trade-off between safety and reliability. To understand this, we need a different way to model reliability and safety. We look at the probabilities of three possible types of responses to a client request to the system:

- The probability of a *good response* p_g, which is as intended based on the problem (not only to the specification, as it could be faulty) and delivers a correct value in a timely manner.
- The probability of a *faulty response* p_f, meaning that the system responded but returned an erroneous or malicious result instead of the intended result (i.e. a *byzanthine fault*, as described later in the subsection on Types of Possible Faults).
- The probability of *no response* p_n, due to a system crash or under other *fail-stop* conditions, as described later.

With these probabilities, we can then have a simplified model for reliability:

$$R \coloneqq 1 - p_n - a \cdot p_f, \quad 0 \leq a \leq 1$$

where a is the fault detection factor, denoting the rate of faulty results detected

by the system (and resulting in a fail-stop, i.e. the faulty results are not returned). A system, where faulty responses are not detected, can be therefore perceived as more reliable (as the user may not be able to dinstinguish faulty and intended results), while this can pose a severe safety threat. Take for example a banking transaction. With a bank transfer, safety is more important than reliability, which means in case of a fault, no money should be transferred at all, instead of sending the wrong amount of money or even to the wrong account (or to the sneaky hacker trying to steal your money).

On the other hand, safety can then be modeled as

$$S \coloneqq 1 - b \cdot p_f - c \cdot p_n, \quad 0 \leq b, c \leq 1$$

where b and c are the probabilities that a faulty and no-response event, respectively, will cause undesired behavior or even disastrous consequences to the user. Oftentimes, we have systems where $b \gg c \approx 0$, which means that the safety depends on correct, as-intended results, such as in the bank transfer example.

This definitions of safety and reliability are related to the definition of the *safety and liveness properties* in the concurrency literature. If a system applies one of these properties, then it is true for every possible execution in the system:

- **Safety**: Something bad will not occur,
- **Liveness**: Something good will *eventually* occur.

2.1.2.1 Types of Possible Faults

> To a first approximation, we can say that accidents are almost always the result of incorrect estimates of the likelihood of one or more things.
>
> — C. Michael Holloway, NASA

On the one hand, faults[4] can be categorized by the nature of their timing:

- **Transient faults**: These faults occur once and then disappear. For example, a network request that timed out but succeeded after a retry.
- **Intermittent faults**: These faults occur many times in an irregular fashion and are hard to repeat reliably. Often, several different events must occur at

[4] A *fault* is the initial root cause, including machine and network problems and software bugs. A *failure* is the loss of a system service due to a fault that is not properly handled [25]. The probability of a fault manifesting itself as a failure is not uniform: only a few faults actually cause a system failure, and the actual system downtime is caused by an even smaller group of faults. When thinking about fault detection, it is therefore important to identify and focus on this small, but significant group of faults.

the same time to contribute to and cause such a fault, so that the fault appears random; as a result, it is more complicated to perform a root cause analysis for such faults.

- **Permanent faults**: These faults are persistent and either make the system halt (fail-stop) or cause ongoing faulty behavior of the system. These faults persist until they are actively fixed, but since they are permanent, their detection is more straightforward than that of intermittent errors.

Studies have shown that intermittent and transient failures cause a significant portion of downtime in large scale distributed systems [30].

On the other hand, we need to categorize faults in such a way that we can decide which and how many faults a system should tolerate and how we want to implement this fault-tolerant behavior while still having a useful and maintainable system. No system can tolerate all kinds of faults (e.g. if all nodes crash permanently), so we need a model that describes the set of faults allowed. There are different types of such faults and two major models to describe them [31]:

Crash Failure/Fail-Stop. Processes with fail-stop behaviour simply "die" on a fault, i.e. they stop participating in the protocol [32]. Such a process stops automatically in response to an internal fault even before the effects of this fault become visible. In asynchronous systems, there is no way to distinguish between a dead process and a merely slow process. In such a system, a process may even appear to have failed because the network connection to it is slow or partitioned. However, designing a system as a fail-stop system helps mitigate long-term faulty behavior and can improve the overall fault-tolerance, performance and usability by making a few assumptions [30]: one approach to assuming such a failure is to send and receive heartbeats and conclude that the absence of heartbeats within a certain period of time means that a process has died. False detections of processes that are thought to be dead but are in fact just slow are therefore possible but acceptable as long as they are reasonable in terms of performance. With a growing number of processes involved in a system, such failure detection approaches can become slower and lead to more false assumptions, so more advanced error detection methods come into play, for example based on gossiping [33] or *unreliable failure detectors* as in the Chandra–Toueg consensus algorithm [34].

Examples for faults causing such a fail-stop failure in a distributed system are Operating System (OS) crashes, application crashes, and hardware crashes. As shown before, intermittent and transient failures, if not handled properly, are responsible

for a huge portion of downtime of systems. Because they can lead to unexpected behavior if not properly detected, most systems are designed to *fail gracefully* in the event of such failures, e.g., through proper code design that always throws a runtime exception that is caught by the application runner and forces a reboot (or other strategies, such as a network reconfiguration in consensus protocols). Candea et al. suggest designing systems to be crash-only in the event of faults [30] as they have shown that a reboot can actually safe time, since faults are oftentimes resolved by a reboot, and the time to reboot (the MTTR) can be shorter than the downtime or slowdown of the system caused by the fault itself. In modern service-oriented approaches, a suitable strategy is to simply "throw away" a faulty node while at the same time reinitialising a new one, rather than directly mitigating the cause of the fault, especially for stateless services. In consensus protocols, shutting down a faulty node and initializing a fresh new node is effective as well because the new node's data can be initialized in the background using snapshotting without affecting the performance of the entire cluster, as we show later in the corresponding section.

Byzantine Faults. Instead of crashing, faulty processes can also send arbitrary or even malicious messages to other processes, containing contradictory or conflicting data. Even a bitwise failure in a network cable can cause such a fault. The effect of the fault becomes visible (while being hard to detect and to distinguish from an intended response) and, if not handled properly, can negatively impact the future behavior of the faulty system, resulting in a *byzantine failure*. Byzantine failures compromise the safety of a system far more, as these faults can result in silent data corruption leaving users with possibly incorrect results. A detector for such faults is difficult to create, especially since some faults can only be detected at all under certain circumstances. There is no generic approach to detect such faults in single-node systems. Even worse, numerous security attacks can be modeled as Byzantine failures, such as censorship, freeloading or misdirection. Systems can be protected with *Byzantine fault tolerance* (BFT) techniques (as we will show later in the subsection on Consensus Protocols) by *masking* a limited number of Byzantine failures. This work focuses on *failure masking*, which makes a service fault-tolerant, while it is also worthwhile for the reader to look at *fault detection and correction* techniques[5].

In a naive approach, under the assumption that all nodes process the same

[5]There is not much literature on the subject of byzantine fault detection. The best known approach is the PeerReview detection algorithm [35], which extends the Chandra–Toueg consensus algorithm mentioned earlier. A very recent approach that outperforms PeerReview for transactional databases is Scalar DL [36]. Due to some limitations of fault detection, especially in terms of safety characteristics,

commands in the same order, a system of n replicas is considered to be fault-tolerant (or *k-resilient*) if no more than k replicas become faulty:

- Fail-stop failure: $n \geq k + 1$, as k nodes can fail and one will still be working,
- Byzantine failure: $n \geq 2k + 1$, as while k nodes generate false replies, the remaining $k + 1$ nodes will still provide a majority vote.

This naive approach does not incorporate timing characteristics in messaging between the nodes of the system and the risk of network partitioning. To ensure that all nodes process the same commands in the same order, they need to reach consensus on which command to execute next. This is explained in detail in the subsection on Consensus Protocols), but here's the short version. n nodes are needed to reach consensus in case of k faulty replicas:

- Fail-stop failure: $n \geq 2k + 1$,
- Byzantine failure: $n \geq 3k + 1$.

This also shows that all fail-stop problems are in the space of byzantine problems, too, therefore, a k-byzantine-fault-tolerant system is automatically k-crash-fault-tolerant.

Note that not all authors agree to the binary model of fail-stop vs. byzantine faults, claiming that the fail-stop model is too simple to be sufficient to model the behavior of many systems, while the byzantine model is too generalistic, thus too far away from practical application, and so is byzantine fault-tolerance hard to achieve. At least two other fault models are subject of the literature: the *fail-stutter* fault model [37] is an attempt to provide a middle ground model between these two extremes which also allows for *performance faults*, and the *silent-fail-stutter* fault model tries to extend it furthermore [38]. Despit their usefulness, both models are not quite popular and research on replication protocols does not take those into account, therefore we won't pay too much attention on them in this work. The relation of all this different fault models is shown in figure 2.4.

2.1.2.2 Partition-Tolerance

In addition to the discussed types of faults, there is also the risk of *network partitioning*. There are various types of network partitions, including complete, partial and simplex partitions. In figure **??**, all three are shown for reference. In a), we

but also scalability, failure masking (i.e., byzantine fault-tolerance via consensus protocols) remains the most common and studied approach.

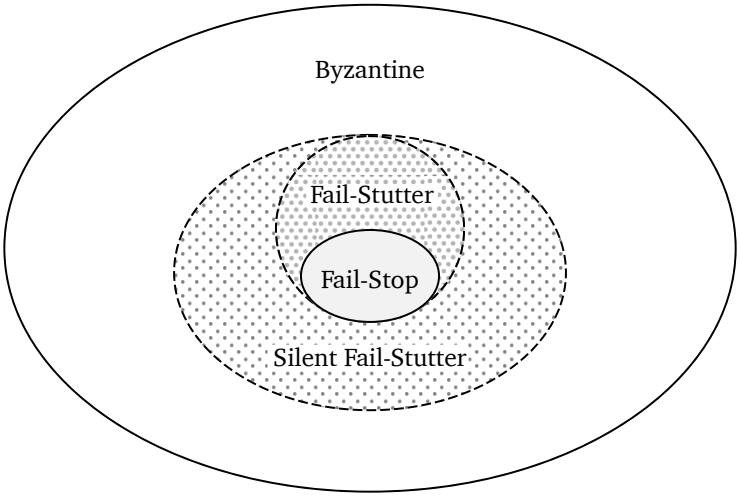


Figure 2.4: Relation of the different fault models

have a complete partitioning of the network, resulting in a split into two disconnected groups, also known as a *split brain*. In b), not all nodes are affected by the partitioning, so there is at least one route between all nodes, which is called *partial partition*. In c), there is a *simplex partition*, in which messages can only flow in one direction. Network partitioning events can be of a temporary or persistent nature, depending on the original fault that caused them, but also on the strategies used to resolve them. In large-scale distributed systems, network partitioning is to be expected. Particularly in massive-scale distributed systems like blockchains, network partitioning is part of the design.

If not handled correctly, the partitioning can lead to Byzantine behavior once the partitioning is resolved again. Partitioning also compromises consistency, as the resulting subnetworks may serve different clients, aggregating different sets of data until they are reunited. Alquraan et al. found that network-partitioning faults lead to silent catastrophic failures, such as data loss, data corruption, data unavailability, and broken locks, while in 21 % of failures, the system remains in a persistent faulty state that persists even after partitioning is resolved [39]. Those faults occur easily and frequently: in 2016, a partitioning happened once every two weeks at Google [40] and in 2011, 70% of the downtime of Microsoft's data centers where caused by network partitioning [41]. Even partial partioning causes a large number of failures (due to bad system design), which may be suprising as there is still a functioning route for messages to pass through the network.

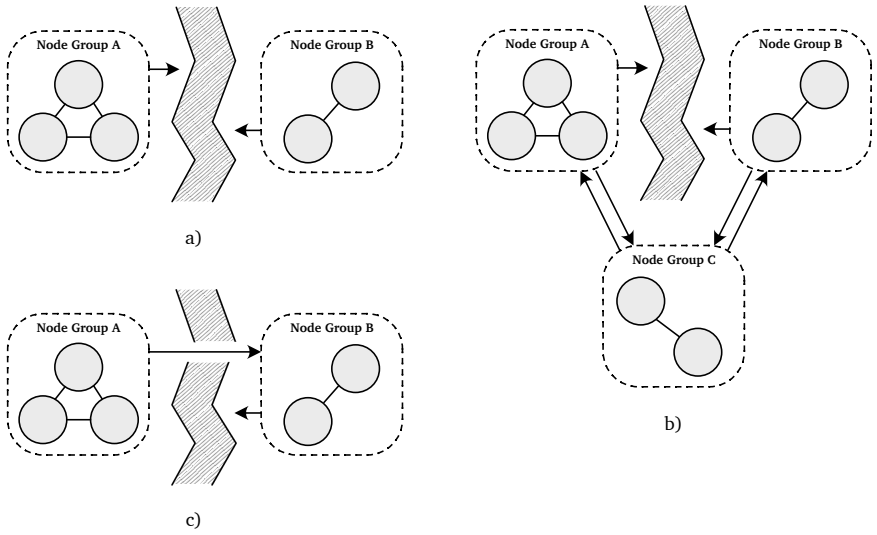


Figure 2.5: Possible network partitioning types. a) Complete partition/split brain, b) partial partition, c) simplex partition.

Partition-tolerance is in general a must-have for a distributed system. In the case of partitioning, users and clients expect the system to still be available and reliable, without compromising the safety by subsequent faulty behavior. The don't want to experience the interruption by the partitioning at all. For a replication protocol to be truly fault-tolerant for large-scale or geographically distributed systems, partitioning must be considered, as it is unavoidable, even partial partitioning as it has been shown. In consensus protocols, this is a mandatory requirement. Container orchestration services like Kubernetes help in resolving partitioning by *network reconfiguration* if they detect node failures and partitioning, which must be taken into account be the consensus protocol.

2.1.2.3 Disaster Recovery

Another type of failure that is important to address are *disasters* in the data center. *Disaster recovery* includes all actions taken when a primary system fails in such a way that it cannot be restored for some time, including the recovery of data and services at a secondary, surviving site. One important disaster recovery measure is *geo-replication*, which can be used to manage disasters of the type that render data centers unavailable due to, for example, a natural disaster such as a flood or earthquake. Geo-replication is discussed briefly in subsection 2.1.6. Since disaster recovery is a complex subject area of its own that goes beyond the application of

replication mechanisms, it is not discussed in detail in this thesis.

2.1.3 High Availability and Dependability

To understand what *availability* means, it is worth reviewing the definitions of availability and discussing them in the context of the other related concepts we have already addressed in the previous subsection, namely reliability and safety. These concepts are summarized under the term *dependability*.

Dependability, as defined by Avizienis et al., can be described in several ways [42]: first, as the ability to provide services that can be justifiably trusted. A more precise definition is also given as the ability to avoid service failures that are more frequent and more severe than is acceptable. In addition to these definitions, dependability is better understood as an integrating concept that subsumes the following requirements:

- **Availability**: readiness for correct service,
- **Reliability**: continuity of correct service,
- **Safety**: absence of catastrophic consequences on the user(s) and the environment,
- **Integrity**: absence of improper system state alterations,
- **Maintainability**: ability to undergo modifications and repairs.

Availability is typically measured by the percentage of time a system is available to users (the total proportion of time a system is operational), while reliability refers to the duration of uninterrupted periods of operation (the MTBF) [43]. For example, a service that goes down for one second every fifteen minutes provides reasonable availability (99.9 %) but very low reliability.

The availability of a system monitored over a time period $[0, t]$ is expressed by dividing the total uptime by the total time of monitoring the system (i.e., the sum of uptime and downtime) [44]:

$$A(t) = \frac{\sum_i u_i}{t}$$

where u_i are the periods of uptime.

In case of fail-stop failures, this can also be simplified and expressed by the Mean Time To Failure (MTTF) (equation 2.1) and Mean Time Between Failure (MTBF), as the MTTF describes the mean time of uninterrupted uptime:

$$\lim_{t\to\infty} A(t) = \frac{\text{MTTF}}{\text{MTBF}} = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

where in this case the MTTR is simplified also includes the MTTD. Note that this won't apply to byzantine failures, as the system is still operating even in case of a failure (which again shows that availability and reliability alone are not sufficient to describe dependability).

Different levels of availability and their respective downtimes are listed in the table 2.1. They are divided into availability classes based on the Availability Environment Classification (AEC) of the Harvard Research Group (HRG). Note that this notation may be outdated due to the ambiguous use of the terms disaster, reliable, available, and fault-tolerant, but is still the most complete and accepted classification scheme.

Table 2.1: Various availability classes and the respective annual downtime, based on the Availability Environment Classification (AEC) of the Harvard Research Group (HRG)

Class	Availability	Annual Downtime	Description
Disaster Tolerant	99.9999 %	31.56 s	Service must be available under all circumstances
Fault Tolerant	99.999 %	5.26 min	Service must be guaranteed without interruption, 24/7 service must be assured
Fault Resilient	99.99 %	52.60 min	Service must be assured without any downtime within well defined time windows or at main runtime
High Availability	99.9 %	8.77 h	Service is only allowed to be interrupted within scheduled time windows or minimal at main runtime
High Reliable	99 %	3.65 d	Service can be interrupted, data integrity must be assured
Conventional	-	-	Service can be interrupted, data integrity is not essential

An availability of 99.999 % is referred to as *five nines* and is the typical standard for telecommunication networks and a hit target for many site reliability engineering (SRE) departments of service providers, while popular commercial platforms provide lower levels of availability, such as Amazon Web Services (AWS), which provides

99.99% availability for their standard offering of S3 [45]. Oftentimes, such services are geo-replicated and deployed in specific *availability zones* so that they are available during peak operating hours for the corresponding time zones. The maintenance time (e.g. for updates and general MTTR) is then usually scheduled for the night hours. For critical business operations, a certain level of availability must be guaranteed by the service providers, which they typically specify in their *Service Level Agreements* (SLAs). If the service provider fails to keep these availability promises, the provider declares themselves responsible to compensate for the damage incurred (the missed revenue). The confidence of having an excellent SLA that ensures high availability, among other things, can often be a decisive competitive advantage. For example, the SLA for AWS S3 guarantees customers tiered compensation based on the availability achieved in the corresponding billing cycle. As of the time of writing, partial reimbursements start when an availability of 99.9% wasn't met [46]. The total cost of the billing cycle will be refunded if the service didn't met an availability of at least 95 % throughout this cycle. At the same time, Microsoft is offering an SLA for its Cosmos DB, a distributed NoSQL database, that guarantees partial reimbursement for certain availability regions even if a five nines availability (99.999 %) is not met, but they also cover reimbursements if latency and consistency guarantees are not met [47].

When orchestrating multiple services in the context of a *service oriented architecture* (SOA), the availability of a resulting aggregated system or business process depends heavily on all the orchestrated services as well as the orchestrating system itself, and decreases for each participating system. Ignoring network availability, overlapping unavailability periods and other factors, and assuming all services are run in a sequential order, we can roughly estimate the overall availability as

$$A_{\text{serial}}(t) \approx \prod_i A(t)_i$$

where $A(t)_i$ are the availabilities of the various systems in the period of t, including the orchestration engine. Note that this is just a naive estimation that does not take into account failover and compensation mechanisms of the orchestration engine that can actually improve the availability again. Figure {fig:serial-availability} illustrates this for two nodes.

Conversely, replication is a way to drastically increase the availability of a system. By having more than one copy of important information, the service continues to be usable even when some copies are inaccessible. By providing a *High-Availability Cluster* (HA Cluster)[6] of replicas, the probability of failure of the whole cluster during

[6] Service load balancing is another technique for achieving high availability through service replication.

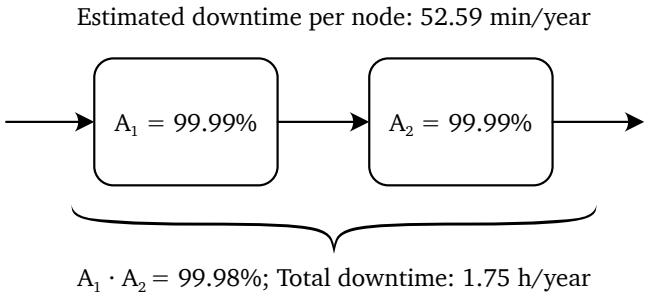


Figure 2.6: Example for the resulting availability of a sequential orchestration of services

a fixed period of time decreases with every replica, decreasing the estimated annual downtime. When we ignore factors like failures of the replication system itself, faults in the underlying network, and the time needed for cluster reconfigurations on node failures (as included in the cluster MTTR), we can model the availability of the cluster naively as

$$A_{\text{cluster}}(t) \approx 1-(1-A(t))^n = 1-U(t)^n$$

where n is the number of replicas, $A(t)$ is the availability of a single node in the period of t, and $U(t)$ the respective unavailability. A replica configuration like this fails if all of its replicas fail. Figure {fig:parallel-availability} illustrates this for two nodes.

As shown, the common way to achieve high availability is through the replication of data in multiple service replicas. High availability comes hand-in-hand with fault-tolerance, as services remain operational in case of failures as clients can be relayed to other working replicas.

2.1.4 Consistency

> A man with a watch knows what time it is. A man with two watches is never sure.
>
> — Segal's law

Building scalable and reliable distributed systems requires a trade-off between consistency and availability. Consistency is a property of the distributed system

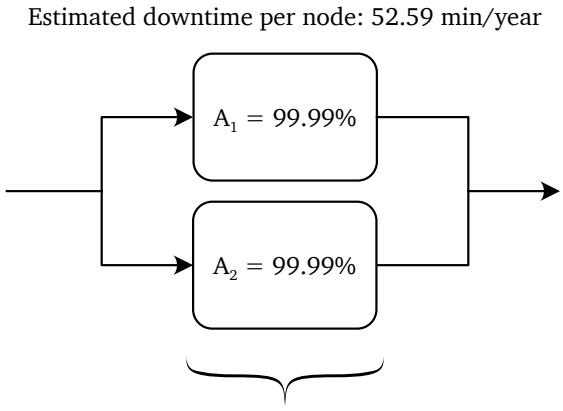


Figure 2.7: Example for the resulting availability of a replica set

that ensures that every node or replica has the same view of the data at a given point in time, regardless of which client updated the data. Ideally, a replicated data store should behave no differently than one running on a single machine, but this often comes at the cost of availability and performance. Deciding to trade some consistency for availability can often lead to dramatic improvements in scalability [48].

Consistency is an ambiguous term in data systems: in the sense of the *ACID model* (Atomic, Consistent, Isolated and Durable), it is a very different property than the one described in the *CAP theorem* (we explain this later in this section). The ACID model describes consistency only in the context of database transactions, whereas consistency in the CAP model refers to a single request/response sequence of operations. We focus on the definition in the CAP theorem in this work. In the distributed systems literature in general, consistency is understood as a spectrum of models with different guarantees and correctness properties, as well as various constraints on performance.

A database consistency model[7] determines the manner and timing in which a

Since this work only deals with data replication, this topic is not covered here (although some session data must be replicated between these services if they are not stateless in the first place).

[7] Originally, consistency models were discussed in the context of concurrency of operations on a single machine or set of CPUs respectively in multi-threaded environments. Discussions of consistency models in distributed systems or even distributed database systems are not substantially different, since a distributed system is basically nothing more than a collection of machines on which multiple threads and processes can run concurrently or in parallel.

successful write or update is reflected in a subsequent read operation[8] of that same value. It describes the ordering guarantees for the execution of operations accessing the system and what values are allowed to be returned [49]. There exists no one-size-fits-all approach: it is difficult to find one model that satisfies all main challenges associated with data consistency [17]. Consistency models describe the trade-offs between concurrency and ordering of operations, and thus between performance and correctness.

We assume a classical memory system at each node whose state is not affected by reads as opposed to a *quantum memory* system whose state might change on reads.

There are two distinct perspectives on consistency models: the data-centric and client-centric perspective, as illustrated in figure 2.8 [50]. The data-centric perspective analyzes consistency from a replica's point of view, where the distributed system synchronizes read and write operations of all processes to ensure correct results. Similarly, the client-centric perspective examines consistency from a client's point of view. From this perspective, it is sufficient for the system to synchronize only the data access operations of the same process, independently of the others, to ensure their consistency. This is justified because common updates are often rare and mostly access private data [51]. It is sufficient to *appear* consistent to clients, while being at least partially inconsistent between cluster nodes. Therefore, client-centric consistency models are in general weaker than data-centric models.

Following, a few of those consistency models are discussed, starting with the model with the strongest consistency guarantees [12]. Throughout the discussion, we use the following notation (also see figure 2.9) to illustrate the behavior of the models by examples:

- $w(x, v)$ denotes a successful write operation of the value v into the variable x.
- $r(x) = v$ denotes a read of x that returns the value v.

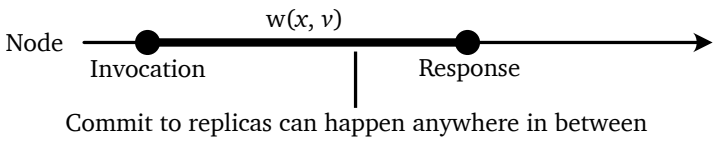


Figure 2.9: Notation of an operation invocation to illustrate following consistency discussions

[8] Throughout this work, the discussed read operations do not change a systems state. There is the concept of a *quantum memory system* whose state can change during reads, which we will not take care of here.

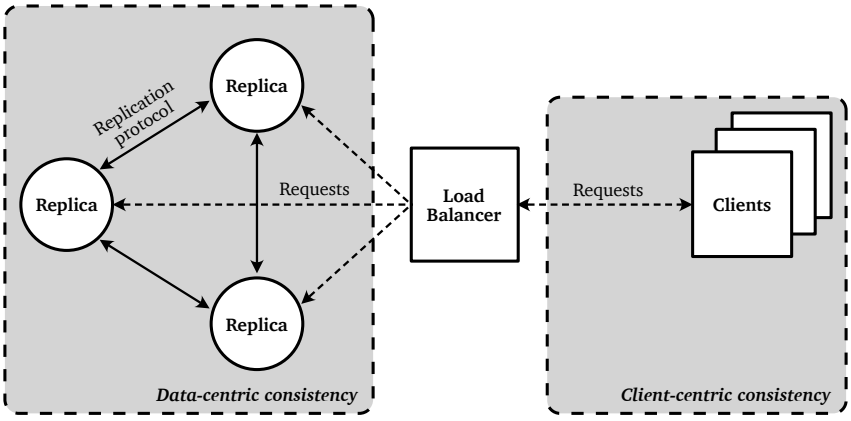


Figure 2.8: The data-centric and client-centric perspective on consistency models, as described by Bermbach et al.

Strong Consistency (Linearizability). Strong consistency is, from the view of a client, the ideal consistency model: a read request made to any of the nodes of the replica cluster should return the same data. A replicated data store with strong consistency therefore behaves indistinguishably from one running on a single machine. A strong consistency model provides the highest degree of consistency by guaranteeing that all operations on replicated data behave as if they were performed atomically on a single copy of the data. After a successful write operation, the written object is immediately accessible from all nodes at the same time: *what you write is what you will read*[9]. This is illustrated in 2.11. A client never sees an uncommitted or partial write. However, this guarantee comes at the cost of lower performance, reliability, and availability compared to weaker consistency models [13], [52]: Algorithms that guarantee strong consistency properties across replicas are more prone to message delays and render the system vulnerable to network partitions, if not handled properly. Such algorithms do need special strategies to provide partition-tolerance to the cost of availability.

In the literature, strong consistency is often referred to as *linearizability*. To achieve linearizability, an absolute global time order must be maintained [53]. Each operation must appear to be executed immediately, exactly once, at some point between its invocation and its response [54]. This is illustrated in the figures 2.12 and 2.13. If a write call returns, the client can be sure that the written data is

[9]For this reason, it is sometimes referred to as *read-after-write consistency* from a client-centric view.

available throughout the system, just as it would be with a single-node system. Therefore, consistent data stores are easy to use for developers because they do not need to be aware of the differences between the respective replicas of the data store.

Strong consistency is mandatory for use cases with zero tolerance for inconsistent states and strong requirements for real-time ordering guarantees, such as

- Business process automation,
- Service orchestration (SOA),
- Financial services,
- Stock exchange trades and analysis,
- Event sourcing,
- Aviation systems.

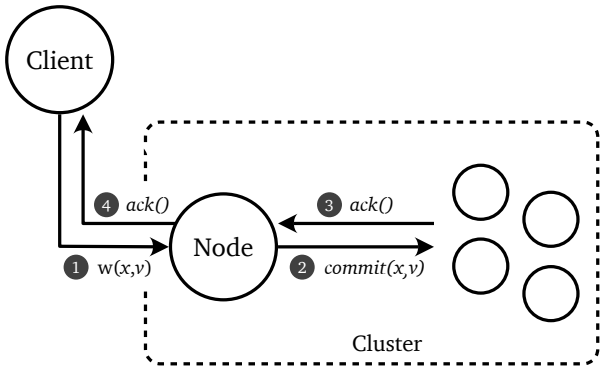


Figure 2.10: Communication model between clients and replica cluster nodes to ensure strong consistency. A write is acknowledged only when it is consistent throughout all cluster nodes, therefore synchronizing the operations.

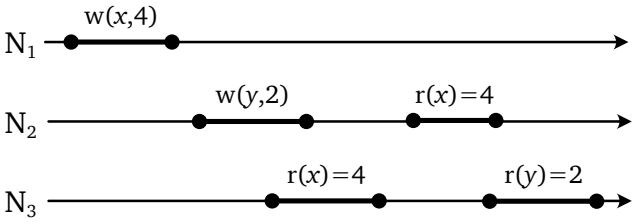


Figure 2.11: Operation schedule that satisfies the real-time ordering guarantee of the strong consistency model (linearizability)

The linearizability constraints can be further hardened, resulting in *strict consistency*. While linearizability takes overlapping operations in a relaxed way, as

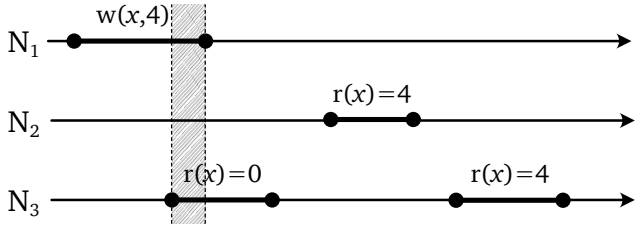


Figure 2.12: A strongly consistent operation schedule with a read-write overlap. During the overlap, the read is allowed to return either the current or the previous write.

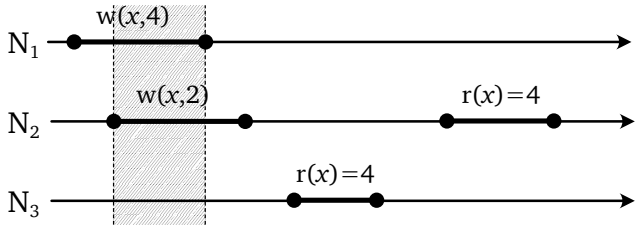


Figure 2.13: This operation schedule is still linearizable, as the order of committing the overlapping writes is not guaranteed.

Table 2.2: Fact sheet for strong consistency

Consistency	Strongest
Ordering guarantees	Real-time (for non-overlapping operations)
Availability	Lowest
Latency	High
Throughput	Lowest
Perspective	Data-centric

illustrated in figure 2.13, strict consistency does not give that freedom. Overlapped operations also need to be ordered in strict real-time order by the time of their invocation. In practice, strict consistency is hard to implement and is rarely necessary for practical use cases, hence it is reduced to a theoretical basis only.

Sequential Consistency. Sequential consistency weakens the constraints of strong consistency[10] by dropping the real-time property. If a write call returns, it must not

[10]It is easy to show that a schedule of operations that satisfies linearizability also satisfies sequential consistency.

neccessarily be available in a subsequent read, but when it is, then the correct order of the write operations of the originating node or process is guaranteed [52]. That is, linearizability takes care of time and sequential consistency takes care of program order only. There are two properties defining sequential consistency: first, the writes of one node must appear in the originally executed order (program order) on every node. Second, the order of writes between nodes is not specified, but all nodes must agree on a sequential order (global order) while ensuring program order. This is illustrated in 2.15.

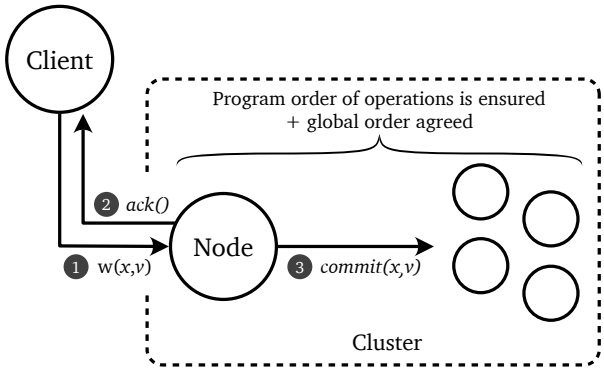


Figure 2.14: Schematic communication model between clients and replica cluster nodes for sequential consistency. A write can be acknowledged before it is propagated consistently across all cluster nodes. For all successful writes that are committed throughout the cluster, the original order is guaranteed.

All reads at all nodes will see the same order of writes to ensure sequential consistency, but not necessarily in the absolute order (by timestamps) in which clients requested the reads, as the latter can often be impractical as it can lead to reordering of operations between nodes when concurrent writes appear in the wrong order. With sequential consistency, programmers must be careful because two successive writes from different nodes (in real time) to the same value can occur in any order, resulting in unexpected overwrites.

Causal Consistency. Causal consistency relaxes the constraints of sequential consistency by removing the requirement for global order[11]. A system is causally consistent if all operations that are *causally dependent* must be seen in the same order on all nodes. All other (concurrent) operations can appear in any arbitrary order at the nodes, as well as the sets of causally related operations, and nodes do not need to

[11]It is easy to show that a schedule of operations that satisfies sequential consistency also satisfies causal consistency.

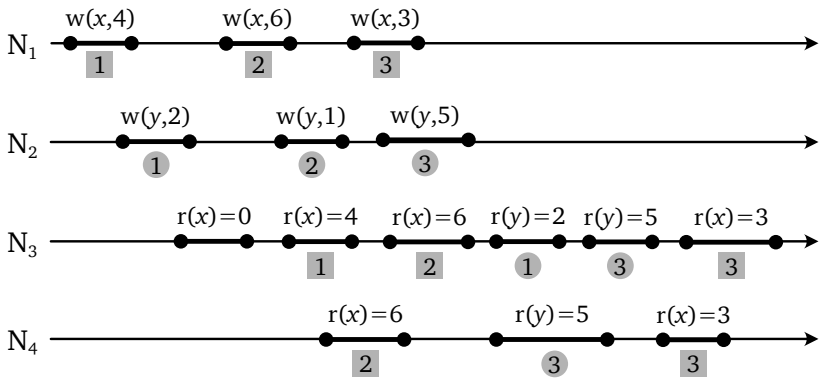


Figure 2.15: Operation schedule that satisfies the global ordering guarantee of the sequential consistency model. The writes of N_1 and N_2 are seen by N_3 in the order of their program execution on the particular nodes, while the order of the interwoven operations does not neccessarily correspond to the real-time sequence. Both N_3 and N_4 have also agreed on one (partial) global order, while N_4 experiences updates faster.

Table 2.3: Fact sheet for sequential consistency

Consistency	Strong
Ordering guarantees	Global ordering
Availability	Low
Latency	High
Throughput	Low
Perspective	Data-centric

agree on a global ordering. Causality is therefore a partial ordering on the set of all operations.

An operation b is causally dependent on an operation a (denoted by $a \rightarrow b$) if one or more of the following conditions are met [53]:

(1) a and b were both triggered on the same node and a was chronologically prior to b.

(2) a is a write, b is a read, and b reads the result of a.

(3) (Transitivity) c is causally dependent on an operation b, which in turn is causally dependent on a: If $a \rightarrow b$ and $b \rightarrow c$ then $a \rightarrow c$.

Two operations a and b are said to be concurrent if $a \not\rightarrow b$ and $b \not\rightarrow a$.

Causal consistency is mostly studied and used in geo-replication (see subsection 2.1.6) and partial replication, as it still satisfies a latency less than the maximum wide-area round-trip delay between replicas [55], and it only cares about a partial ordering that is of interest for the client or user: users are often only interested in a (causally related) subset of events, e.g., those that happened close to their location. An example to illustrate this are social media posts: when a user posts a status update and another user reads and replies to that update, there is a causal order on the two updates, and they should appear in that order to other (subscribed) users. However, when other users send totally unrelated updates, the order in that these updates appear is not important (at least from a consistency point of view). Another example are stock markets: operations on a a single stock (as a reaction to a stock value change) must be consistently ordered, while changes across different, independent stocks ca be seen in different orders.

There are some extensions to harden causal consistency slightly, namely *causal+* (or *causal consistency with convergent conflict handling*), which leverages the existence of multiple replicas to distribute the load of read requests [56], and real-time causal consistency (RTC).

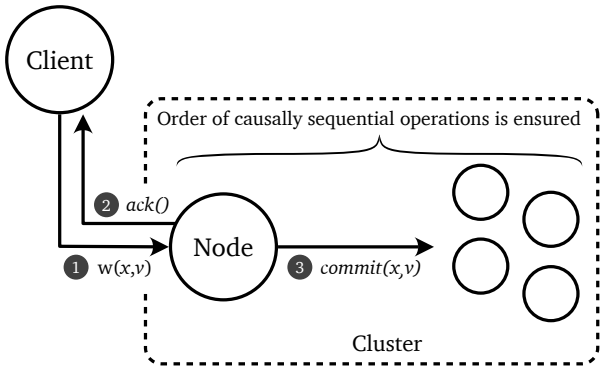


Figure 2.16: Schematic communication model between clients and replica cluster nodes for causal consistency. A write can be acknowledged before it is propagated across all cluster nodes. For all causally sequential writes that are committed throughout the cluster, the original order is guaranteed.

Eventual Consistency. To achieve a higher level of consistency, synchronization between replicas is usually required, increasing the latency and even rendering the system unavailable if network connections between the replicas fail. For this reason, modern replicated systems that put emphasis on throughput and latency often forgo synchronization altogether; such systems are commonly referred to as *eventually*

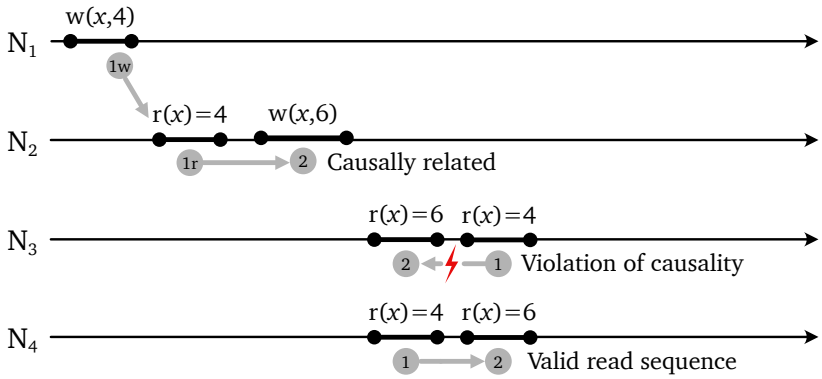


Figure 2.17: Operation schedule that violates causal consistency. As the writes of N_1 has already been observed by N_2 before overwriting it (assuming that its observation may have triggered the overwrite, thus they are causally related), it must be seen in this order by all subsequent reads of all nodes. Since N_4 reads the most recent (the overwriting) value before the overwritten one, it ignores the causal relation and thus violating the properties of causal consistency.

Table 2.4: Fact sheet for causal consistency

Consistency	Moderate
Ordering guarantees	Causal
Availability	High
Latency	Moderate
Throughput	Moderate
Perspective	Data-centric

consistent [57]. Eventual consistency is a weak consistency model that does not guarantee any global ordering, but only *liveness*: intermediate states are allowed to be inconsistent, but after some time, in the absence of updates, all nodes should converge, returning the same resulting state set of operations [58]. This is illustrated in 2.20. Unlike strong consistency models that maintain the illusion of a single-site system, weaker models allow clients to see that the system has replicas. As illustrated in figure 2.19, in eventually consistent distributed databases, a replica performs an operation requested by a client locally without any synchronisation with other replicas and immediately acknowledges the client of the response. The operation is passed asynchronously to the other replicas and, in the case of network partitioning, can be pending for a while. The time taken by the replicas to get consistent may or

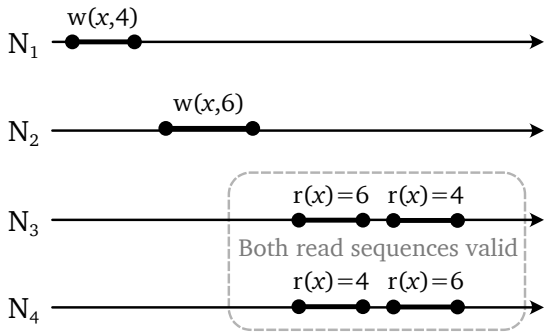


Figure 2.18: Operation schedule that meets the requirements for causal consistency. Since there is no causal relation between the writes of N_1 and N_2, any order of occurrence of the values in subsequent reads satisfies the properties of causal consistency.

may not be defined, but the model clearly requires that in the absence of updates, all replicas converge toward identical copies. This often requires *conflict resolution* techniques.

Eventual consistency could be somehow referred to as the incarnation of the liveness property described in subsection 2.1.2 ("something good will eventual happen"), while safety is not neccessarily guaranteed: it depends on the use case and potential conflict resolution.

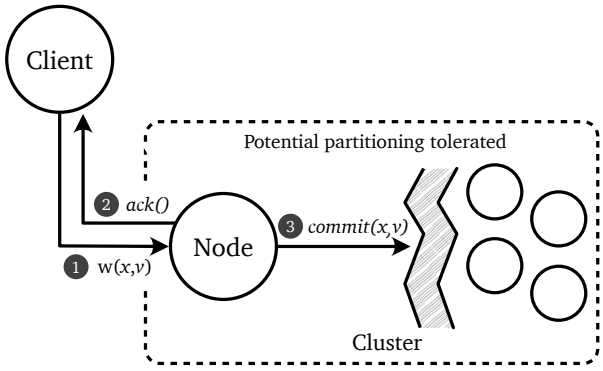


Figure 2.19: Schematic communication model between clients and replica cluster nodes for eventual consistency. A write is acknowledged immediately before it is propagated across all cluster nodes. The system remains available even when partitioned by allowing disconnected nodes to converge later when reconnected again.

Because of the possibility of inconsistency, application developers integrating eventually consistent data stores must be explicitly aware of the replicated nature of data items in the store. Compared to strong consistency, the developer must

consider the asynchronous behavior and take care that stale data does not render their application unusable, and also make their end users aware of this through clear in-application communication. We describe these considerations in detail in the subsection 2.1.8.1.

There are also use cases where eventual consistency is ideal since they do not require any ordering guarantees, for example:

- Non-threaded comments, reviews or ratings,
- Incremental counters, such as of likes in social media or views on videos.

One of the best known examples of an eventually consistent distributed system is the *Domain Name System* (DNS). DNS is a hierarchical, highly available system that handles billions of queries every day. The values are cached and replicated across many servers, while it takes some time for the update of a particular record to propagate through DNS servers and clients.

In the database literature, eventually consistent databases are often classified as *BASE* (Basically Available, Soft-state, Eventually consistent), as opposed to the traditional ACID requirements for *relational database management systems* (RDBMS) [48]. *Basically available* describes that services are available as much as possible, but data may not be consistent. *Soft-state* describes the convergence behavior, that after a certain amount of time, there is only a certain probability of knowing the actual state. In the past, NoSQL databases were often implemented to meet BASE requirements, hence being eventually consistent. This has changed over time, such as MongoDB supporting strong consistency and also ACID transactions.

Eventual consistency can become a problem when operations aren't idempotent and users repeat operations, because they don't receive on a read what they've just written. To mitigate this, another consistency model has been derived from eventual consistency, called *strong eventual consistency*. It enforces *idempotency* and *commutativeness* on all operations. By that, strong eventual consistency hardens the liveness property by adding a safety guarantee to the model (see subsection 2.1.2 for reference): any nodes that have received the same (probably unordered) intermediate set of updates will be in the same state. An example of a use case for strong eventual consistency are *conflict-free replicated data types* (CRDT), which are described in subsection 2.1.9.6.

Weak Consistency. As its name indicates, weak consistency offers the lowest possible ordering guarantee, since it allows data to be written across multiple nodes and always returns the version that the system first finds. This means that

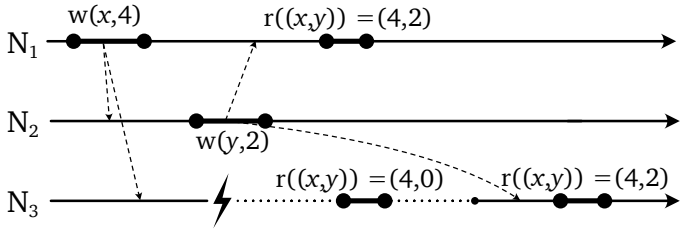


Figure 2.20: Operation schedule for eventual consistency. One node is partitioned from the others, returning an inconsistent, outdated state in between. When reconnected, the state of the partition converges and eventually becomes consistent again.

Table 2.5: Fact sheet for eventual consistency

Consistency	Lowest
Ordering guarantees	None
Availability	High to Highest
Latency	Low
Throughput	Highest
Perspective	Client-centric

there is no guarantee that the system will eventually become consistent. Only writes that are explicitly synchronized are consistent. Everything else in between is unordered, but at least the same set of operations on all nodes. This requires programmers to explicitly synchronize operations. Synchronized operations are sequentially consistent, as they are seen by all processes in the same order. Weak consistency with explicit synchronization is uncommon in distributed systems, but a common model in concurrent, multi-threaded programming.

So far, this subsection has briefly introduced consistency models[12]. Table 2.7 summarizes all the discussed consistency models. The next subsection explains how tradeoffs are made between consistency, availability, and latency, and how designers of distributed database systems can choose a consistency model (or multiple models) that meets the needs of their applications.

[12]In addition to the models discussed in this work, there are other consistency models. Since they do not play an important role for the kind of systems discussed in this work, we will not investigate them further, but provide a uncommented list of some of them: session consistency (e.g. monotonic read, monotonic write, read-my-write, write follows read), bounded staleness, consistent prefix and more.

Table 2.6: Fact sheet for weak consistency

Consistency	Lowest to None
Ordering guarantees	Only explicitly synchronized operations (sequential), else none
Availability	High
Latency	Low to Lowest
Throughput	Highest
Perspective	Data-centric

Table 2.7: Consistency models in descending order of strictness and at the same time in ascending order of availability

Strictness & Ordering Guarantees ↑

	Data-centric
Strong consistency	After a successful write operation, the value can be read at all nodes immediately
Sequential consistency	No immediate read, but writes have to be seen in the same order at every node
Causal consistency	Only causally related writes must be seen in the same order at every node
Weak consistency	Only writes that are explicitly synchronized are consistent. Everything else in between is unordered
	Client-centric
Eventual consistency	After the last update, all replicas all replicas converge toward identical states, eventually becoming consistent, but with no time and ordering guarantees

↓ Availability & Throughput

2.1.4.1 The CAP Theorem

Why can't we always have strong consistency? The CAP theorem (CAP stands for Consistency, Availability and Partition Tolerance), also known as Brewer's Theorem, named after its original author Eric Brewer [59], asserts that the requirements for

strong consistency and high availability cannot be met at the same time and serves as a simplifying explanatory model for consistency decisions that depend on the requirements for the availability of the distributed system in question [60].

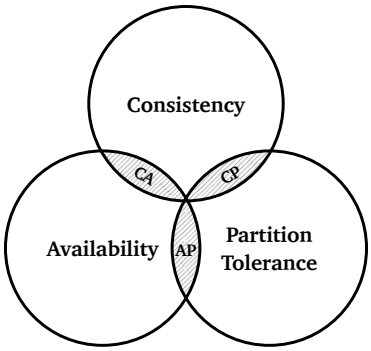


Figure 2.21: Illustration of the CAP theorem

In this chapter, the three properties of the CAP theorem have already been presented. We summarize them here in the context of this theorem:

- **(Strong) Consistency**: The system satisfies linearizability, thus all clients see the same data at the same time.
- **Availability**: The system operates even in case of node failures (is fault-tolerant), so read and write requests always receive a response.
- **Partition Tolerance**: The system continues to work even under arbitrary network partitioning. This is a mandatory requirement in distributed systems.

The theorem states that only two of those properties can apply at the same time. This results in the following classes:

- **CA**: A network problem might render the system unavailable. Not suitable for any distributed system. Traditional RDBMS fall under this class.
- **CP**: Strong consistency is guarenteed even in the case of network partitions, but some data may become unavailable.
- **AP**: The system is available even in the case of network partitions, but potentially returning inconsistent data.

This classes reduce the trade-off between availability and consistency to strict binary terms, when in fact, as the various consistency models show, the trade-off is gradual in nature.

In general, partition tolerance should always be guaranteed in distributed systems (as described in section 2.1.2.1). Therefore, the trade-off is to be made between consistency and availability in the case of a network partition. Eric Brewer revisited his thoughts on the theorem, stating that trade-offs are possible for all three dimensions of the theorem: by explicitly handling partitions, both consistency and availability can be optimized [61]. This gives credit to the gradual nature of the trade-off in consistency models.

The theorem is often interpreted as a proof that eventually consistent databases have better availability properties than strongly consistent databases. There is sound critic that in practice this is only true under certain circumstances: the reasoning for consistency trade-offs in practical systems should be made more carefully and strong consistency can be reached in more real-world applications then the CAP theorem would allow [62]. In the next two subsections, we present two critics or extensions to the CAP theorem.

2.1.4.2 The PACELC Theorem

The author of the PACELC theorem criticizes the CAP theorem to just focus on failures [63]. They described the PACELC theorem to extend the CAP theorem by another trade-off in the case of normally operating systems with no partitioning failure: In the case of network partitioning (P), the CAP rule still applies, where one must choose between availability (A) and consistency (C). But else (E), when there are no network partitions and the system is operating normally, one still has to choose between latency (L) and consistency (C).

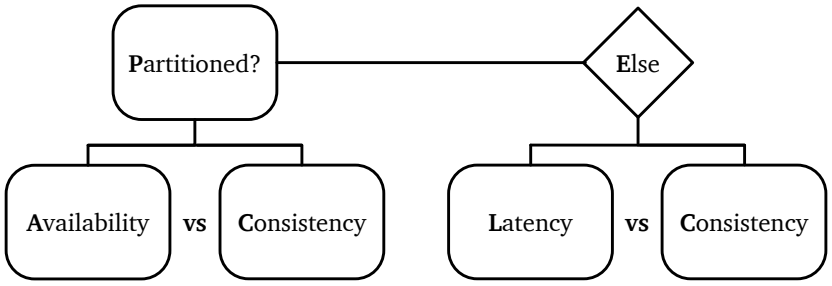


Figure 2.22: Illustration of the PACELC theorem, showcasing the trade-offs in the case of partitioning and also in the absence of network partitioning

The author introduced this theorem to support decision making in the design and implementation of a distributed database system, since consistency decisions must also be made outside of network partitions: the trade-off between consistency and

latency is often even more important and arises as soon as a distributed database system introduces replication.

2.1.4.3 The CALM Theorem

The CALM theorem (Consistency As Logical Monotonicity) arose from a critique of the CAP theorem. It helps to understand whether a distributed problem can be solved with a strong or weak consistency model. While a strong consistency model always requires some form of coordination between nodes which increases latency, a weak model such as eventual consistency can eschew coordination entirely, but often at the cost of violating correctness properties. The CALM theorem allows the system designer to decide whether a weak consistency model can be applied without compromising correctness by considering possible monotonic properties. The theorem says "a problem has a consistent, coordination-free distributed implementation if and only if it is monotonic [64]".

Monotonicity is defined as follows: A problem P is monotonic if for any input sets S, T where $S \subseteq T$, $P(S) \subseteq P(T)$. Figure 2.23 illustrates this using a simple function example.

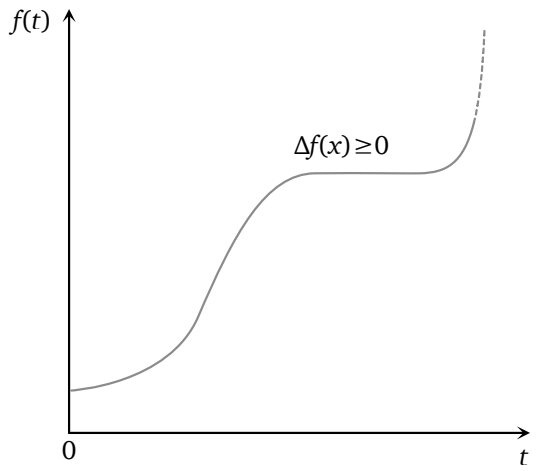


Figure 2.23: A curve for a nondecreasing monotonic function

For a distributed problem to be monotonic, all operations must be designed in a way that they are

(1) Associative: $a \circ (b \circ c) = (a \circ b) \circ c$,

(2) Commutative: $a \circ b = b \circ a$,

(3) Idempotent: $a \circ a = a$.

The challenges of the theorem are

- to determine whether a problem can be described by a monotonic specification at all,
- and, if this is the case, to design and practically implement this specification.

One way to achieve this behavior is to apply the *derived monotonic state pattern* [65]. The pattern is illustrated in figure 2.26, as initially described by Braun. The pattern can be applied by factoring out every operation on the state object into *immutable aggregates*, such as *domain events*. We will illustrate this with the example of a shopping cart. Domain events in a shopping cart are the insert and removal of items, denoted as `itemInserted` and `itemRemoved`. In a naive implementation, both events could be modeled as operations on a mutable state (the shopping cart entity). But with a weak consistency model, the order of these operations can not be guaranteed, as illustrated in figure **??**.

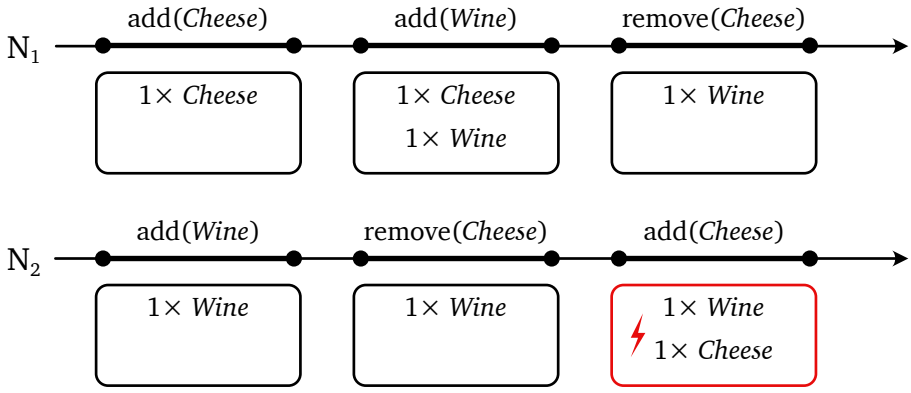


Figure 2.24: Naive implementation of a shopping cart with weak consistency. N_2 received the operations in a different order than N_1, rendering a wrong final state.

But this problem can be implemented in a monotonic fashion: instead of applying the domain events on a single mutable state (an *activity aggregate*), they could just be persisted as immutable aggregates in an append-only manner, resulting in two separate monotonically growing sets. Once the final state needs to be observed, it can be derived from the domain events by putting them together on demand[13].

[13]This is also how modern state management systems like MobX or Redux (that are heavily used in frontend development) do this to ease complex implementation of concurrency problems on single devices by applying *functional reactive programming* paradigms.

The resulting state then describes a derived aggregate, expressed by the count of `itemInserted` minus the count of `itemRemoved` for each particular item.

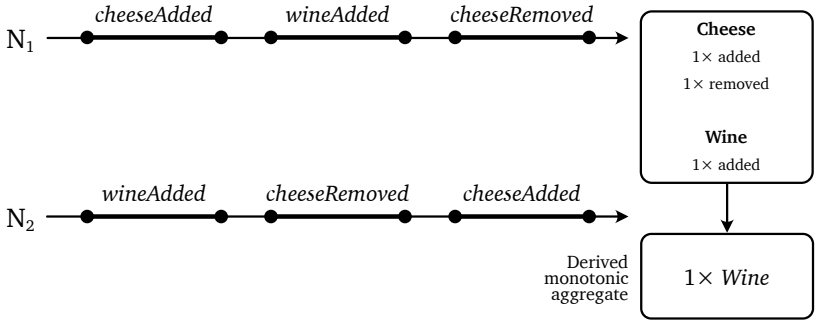


Figure 2.25: Monotonic implementation of the shopping cart problem. Both nodes will derive the same final state without the need for coordination.

Unfortunately, this monotonic behavior is not applicable to all types of operations. There are events that are naturally causally dependent on previous events. In our shopping cart example, this could be a checkout[14]: neither the add nor delete operation commutes with a final checkout operation. If a `checkout` operation message arrives at a node before some insertions of the `itemInserted` or `itemRemoved` events, those events will be lost. We illustrate this in figure 3.9 of chapter 3.

The monotonic property of a problem means that the order of operations does not matter at all. Consequently, in the case of network partitions, both consistency and availability are possible in a monotonic problem, since replicas will always converge to an identical state on all nodes when the partition heals, and this without the need for any conflict resolution or coordination mechanisms.

2.1.5 Partitioning and Sharding

While replication increases the dependability of a distributed system and also helps to reduce latency in geo-replicated systems (described in more detail in subsection 2.1.6), *partitioning* is neccessary to scale out, i.e. to distribute the workload across multiple nodes. Partitioning is the method of breaking a large dataset into smaller subsets. For distributed systems serving many different clients or users, partitioning is essential to maintain both the performance and availability of the system at a high level. With partitioning, the number of nodes of a system grows with the number of

[14]In *event sourcing*, monotonic characteristics can be useful. However, to be able to do *time travel queries*, strong consistency and real-time properties are needed again to derive any intermediate state, at least for all causally related events.

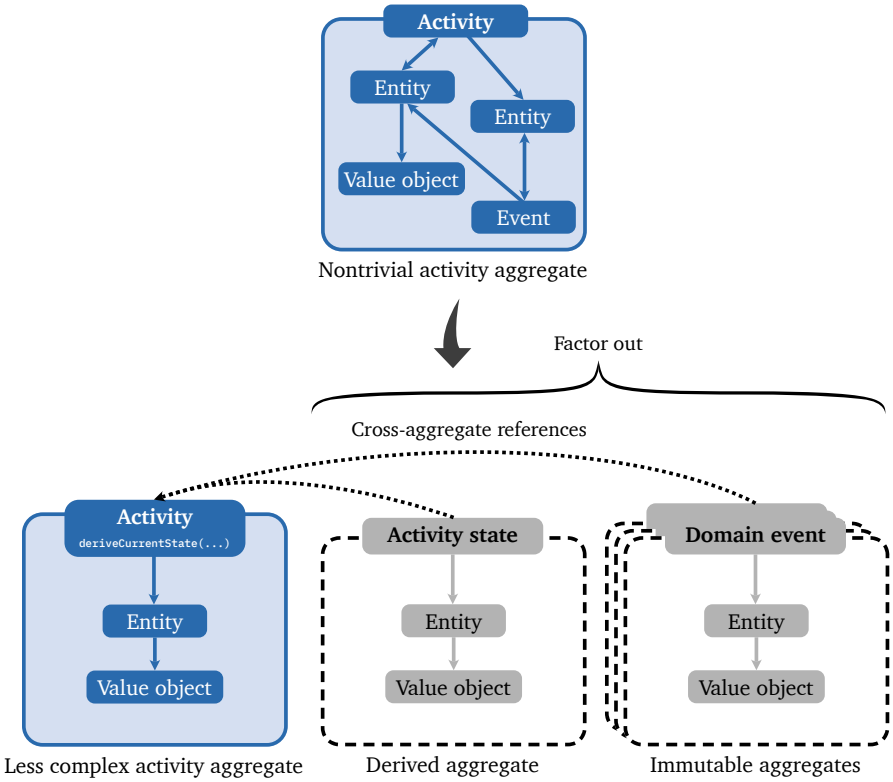


Figure 2.26: Factoring out a nontrivial activity aggregate into immutable and derived aggregates to acquire a monotonic state aggregate allows for a weaker consistency model and therefore lower latency without actually compromising consistency, according to the CALM theorem.

clients. As described in subsection 2.1.1, this happens in general in a linear fashion.

Partitioning helps to improve the performance of both writes and reads: in partitioned databases, when queries only access a fraction of the data that resides in a subset of the partitions, they can run faster because there is less data to scan. This reduces the overall response time to read and load data. Data partitions can also be stored in separate file systems or hardware with different characteristics, depending on the read or write requirements for the content. Thus, data that is accessed very frequently can be held in in-memory caches or fast SSDs, while on the other hand, very infrequent accesses can be stored on dedicated archived mass storage.

There are two ways to partition data: Vertical and horizontal partitioning. Both techniques can be combined. They are described in the following paragraphs and

their differences are illustrated in figure 2.27.

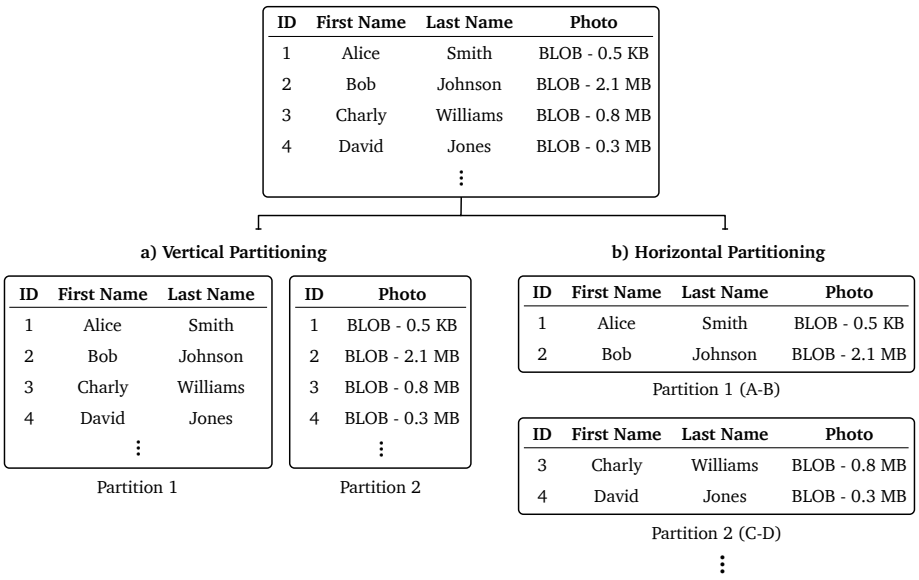


Figure 2.27: a) Vertical vs b) horizontal partitioning, illustrated using a relational database table. In a), the table is split by attributes. In b), the table is split by records, using a lexicographical grouping.

Vertical Partitioning. In vertical partitioning, data collections[15] such as tables are partitioned by attributes. A collection is partitioned into multiple collections, each containing a subset of the attributes. An example of this are database systems where large data blobs that are rarely queried are stored separately (i.e., only when accessing the full set of details for a single record, but not when listing multiple records). Database normalization is also an example of vertical partitioning. Vertical partitioning comes in particularly handy when the partitioned data can be stored in separate file systems or hardware with different characteristics, depending on the read or write frequency and requirements of the different record contents. These requirements may also include dependability and consistency, so that, for example, less critical data may be given an eventual consistency model and lower availability guarantees. On the downside, vertical partitioning can make querying more complicated, so partitioning decisions must be made with proper justification

[15]We will use the term *data collection* to describe structured data as well as semi-structured data that belongs to a certain schema (that describes the structure of the data collection) in any kind of data stores: tables in relational databases, streams in event stores, topics in message brokers, or buckets in file storage systems, to mention a few.

and based on usage estimates.

The need for vertical partitioning can also be reduced by better upfront data design, such as splitting data in trivial facts and deriving an aggregated state instead of managing the a state by continously updating a single data record (cf. subsection 2.1.4.3).

In general, vertically partitioned data is not distributed across multiple nodes, as this would slow down queries: The chances that partitioned attributes of a single dataset will be requested in a single query are high. When partitions are distributed, partitioning strategies should be defined so that cross-partition queries are rare. For this reason, we will not discuss this issue further in this work.

Horizontal Partitioning. In horizontal partitioning, data collections are partitioned by records, while all partitions of the same collection share the same schema. The collection is split by one or more grouping criteria, such as a hash, a certain attribute value or by ranges of attribute values (e.g., by quantitative attributes, date periods, groups of customers or lexicographic ordering). Which split criteria to choose heavily depends on both the use case and technical considerations. With horizontal partitioning, the workload can be distributed across multiple indexes on the same logical server. By choosing well-justified partitioning criteria, frequently visited indexes can be kept small to increase transaction and query throughput.

Sharding. Sharding goes beyond partitioning tables on a single machine by distributing horizontal partitions across multiple machines [66]. This allows to distribute the transaction and query load across multiple servers (both logical or physical).

The optimal *shard key* (resulting from the split criteria in the context of sharding) allows for *load balancing*, i.e. it distributes the workload as evenly as possible across the shards. It also maximizes coherence and locality within a partition and reduces the number of queries and transactions across partitions. For instance, the MongoDB authors recommend choosing shard keys with a high level of randomness for write scaling and high locality for range queries [67]. Good partitioning moves computation close to the node where that data resides, and the master replica of a shard close to where the most clients will write to it (in case of replication with a strong leader, see subsection 2.1.9). Such sharding strategies can also support geographical scalability, as shown in subsection 2.1.6. It also takes the access frequency and importance of every partition into account, so that database users can specify different strategies for security, access permissions, management, monitoring,

and backups of the individual partitions.

To find such a shard key, the distribution of operations over the data collection should be considered. That is, the split should not result in a heavy workload on some partitions while other partitions have a modest workload. A common approach to approximate this—assuming that the estimated workload is evenly distributed—is to split so that each shard contains a similar amount of data. For example, using the first letter of a customer's name results in an uneven distribution because some letters are more common than others. Instead, using a hash of the record ID helps distributing the data more evenly across the partitions.

When querying or writing to the data store, the shard keys for each record to be written and each cursor resulting from the query are calculated and used to look up the corresponding partitions.

Different horizontal partitions can also be served with different dependability and consistency requirements: e.g., data from customers who pay for a better SLA can be provided with higher availability guarantees than data from freemium users.

Rebalancing. Over time, the content and form of a data collection may change, and the distribution of data in the shards may deviate from the distribution originally assumed, especially if *scheme evolution* occurs. In addition to that, the network of the sharded application itself could be reconfigured, for example, by adding data centers or upscaling machines, or in response to a failure or even as a temporary mitigation of a disaster that renders an entire data center unavailable. When this happens, the load on the shards is no longer balanced, resulting in overloaded shards that become slower, which in turn leads to an overall increase in latency and lower availability. To mitigate this, a sharded system must be continuously monitored and *rebalanced* in response to such an event or even proactively when a specific trend is apparent. Rebalancing should ideally be done in such a way that users do not notice it, meaning it should not slow down the system or even reduce its availability [68]: rebalancing should not be part of the MTTR (Mean Time To Repair; for reference, see subsection 2.1.3). Avoiding shard keys closely coupled to actual record values and instead using consistent hashing algorithms to generate sharding keys generally allow for easier rebalancing [69].

Partitioning and Replication. Neither replication or partitioning alone can make a system truly scalable and available. But if both techniques are applied, distributed databases can scale with their users, the data, and the operations executed on it, they can provide world-wide high availability, and they can even withstand data center

outages. In common replicated and partioned architectures, the system is divided into *availability groups*. An availability group consists of a set of user databases that fail over together. It includes a single set of primary databases and multiple sets of replicas. Such groups are often deployed in multiple data centers in different geographic *availability zones* to provide additional disaster resilience. Figure 2.28 illustrates this architecture.

Load balancing becomes more challenging because replicas must be considered when distributing the load among the nodes. In large, horizontally scaled distributed systems, the *replication factor* (the number of replicas per shard) is several magnitudes smaller than the number of total nodes. The load balancer (if it is a centralized agent) or the load balancing protocol (if the load balancer itself is decentralized) must maintain a distribution of replicas that balances the load on the nodes, which means that replicas can be moved between nodes during rebalancing.

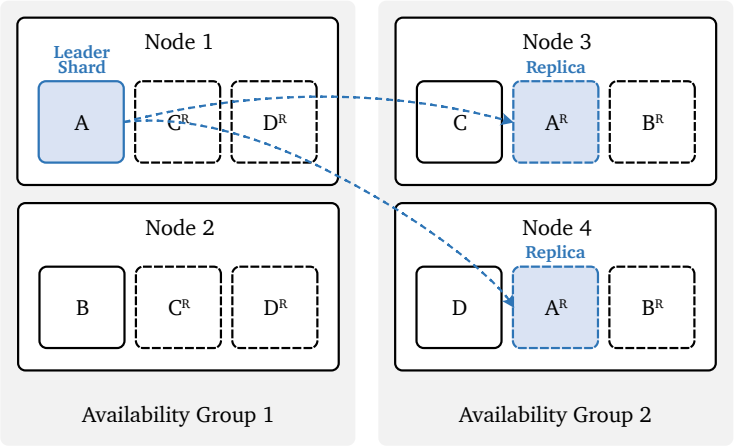


Figure 2.28: Simplified representation of a common replication and partitioning scheme for high availability (with a replication factor of 3). Shards are placed in availability groups (data centers) that are close to their most frequent writers. They are replicated across availability groups so that they can be read with low latency and remain available even in the event of a data center outage.

Transactions on Partitioned and Replicated Databases. Not only the replicated data within a partition, but also the transactions across partitions must follow a consistency model that meets the requirements of the system. In addition to replication between replicas of a shard, transactions must also be ordered and the atomicity of each transaction must be guaranteed. In general, transactions should be linearizable. Following the atomicity property of the ACID schema, every transaction

should be applied to all shards it affects, or none at all.

Providing consistency and atomicity in the execution of transactions in a partitioned and replicated database system is a substantially greater challenge than simply arranging operations in a single replica group, since servers in different shards do not see the same set of operations but must still ensure that they execute cross-shard transactions in a consistent order.

Existing systems generally achieve this using a multi-layer approach, as illustrated in figure 2.29. A replication protocol is used to provide fault-tolerance and dependability inside of a replica group of a shard. Across shards, an commitment protocol provides atomicity, such as the *two-phase commit* (2PC[16]), which is combined with a concurrency control protocol for isolation, e.g. two-phase locking [71]. With geo-replication, even another layer can be added on top to mirror the whole system into multiple geographical regions. Each of this layers adds its own coordination overhead, increasing the number of network round trips significantly and therefore the resulting latency, as illustrated in figure 2.30. NoSQL database systems therefore often forgo transactions altogether to improve availability and latency (see BASE properties of eventual consistent systems in subsection 2.1.4), while NewSQL database systems again allow for transactions and provide ACID properties—mitigating the coordination problem through coordination-free replication and transactions across shards, as shown in subsection 2.1.9.7.

2.1.6 Geo-Replication

The replication techniques discussed so far in this work describe the replication of data within single clusters and data centers to improve the dependability and fault-tolerance of the system. Additionally, to reduce the latency when accessing the system and data from different geographical regions and to hedge against geographically limited disasters, data can be further replicated across clusters in different geographical regions, which is referred to as geo-replication or *cross-cluster replication*. There are several strategies for geo-replication, such as using different replication protocols on multiple layers for intra and inter-cluster replication including asynchronous *data mirroring* techniques or just extending the intra-cluster replication protocol across clusters.

[16]The two-phase commit protocol ensures that all participants in a transaction agreed to run all the operations in the transaction, or none. In the first phase of the protocol (the voting phase), the approval or rejection of the commitment of the changes from all participants is collected. If all participants agree, they will be notified of the result (the commit phase) and all partial transactions will be executed (and ressource locks lifted); otherwhise, the transaction will be rolled back. Thus, since all members must be reachable for a transaction to work, 2PC is also referred to as an “anti-availability” protocol [70].

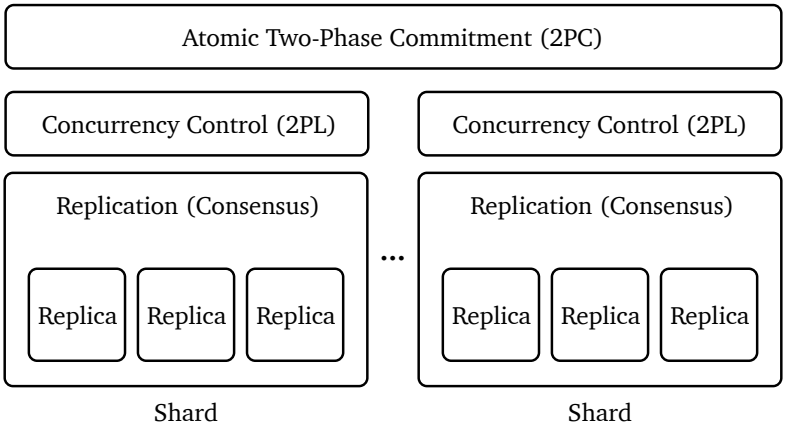


Figure 2.29: Common architecture for a partitioned and replicated data store

Disaster Resilience. Data center failure is one of the most threatening events, as it results in all systems in the data center becoming unavailable; in the worst case, there is serious data loss. Severe outages are not rare—in a recent 2021 survey on data center outages, 6% of respondents reported severe outages in the past year [72].

Many cloud vendors therefore recommend to run your data stores and services on at least two different *availability zones*. This eliminates the risk of a single data center to be a single point of failure, and the risk of unavailability and locally limited faults is significantly reduced.

In a globally distributed database system, there is a direct correlation between the level of consistency and the durability of data in the event of a region-wide outage.

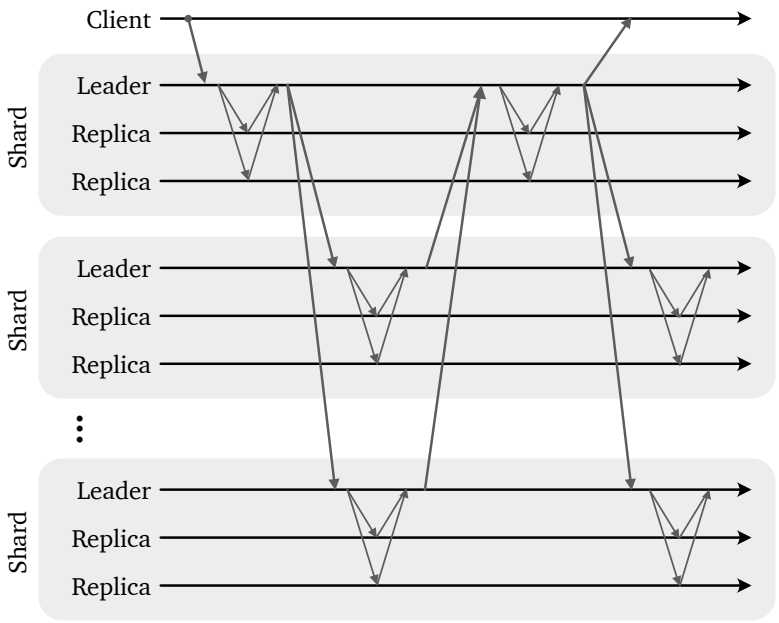


Figure 2.30: Coordination between shards and replicas for consistent transactions with traditional two-phase commits (2PC) and replication

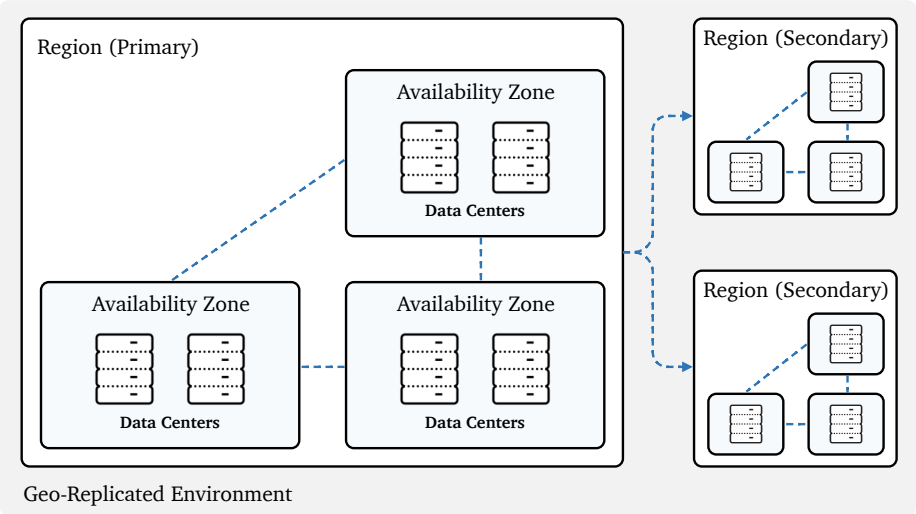


Figure 2.31: Geo-replication scheme common among cloud infrastructure providers: to provide disaster resilience, data is replicated to multiple availability zones in a single region. In addition, they can be mirrored to secondary regions to provide data locality for lower read latency, as well as resilience in the event of large-scale disasters.

Reduced Read Latency. When it comes to read latency, it is recommended to place read replicas in close geographic proximity to clients to ensure *data locality*. This reduces the round-trip time for requests and thus the read latency for clients. Figure 2.32 illustrates this.

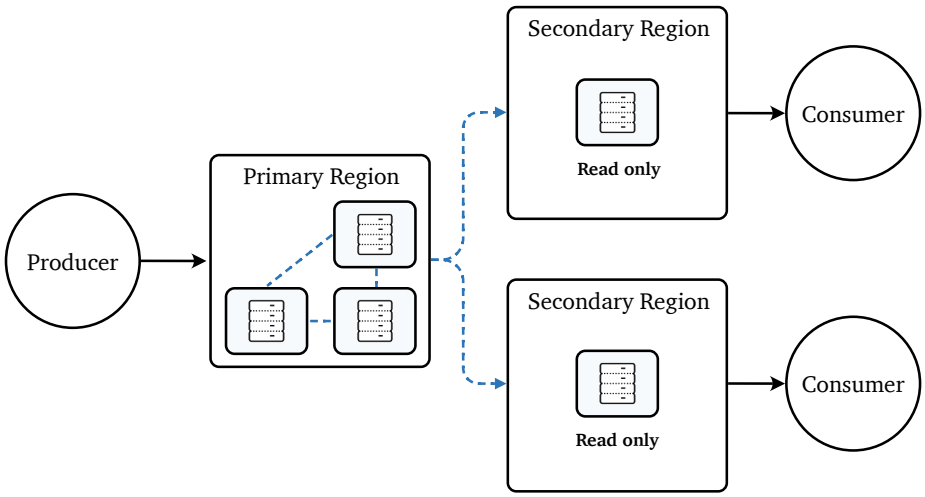


Figure 2.32: By mirroring data into read-only replicas in different regions, data locality is increased and read latency can be reduced

Increased Write Latency. In the strong consistency model, extending the intra-cluster replication protocol across clusters can be very expensive, as more nodes must take part in coordination to reach consensus. Replicating across wide-area networks adds a minimum round-trip time to every request, which can be tens of milliseconds or more across continents. The lower bound of possible latency optimization here is defined by the speed of light: for example, the minimum round-trip to coordinate replicas on AWS data centers across the Atlantic from Dublin, Ireland to North Virginia, USA ($\sim$ 11.000 km round-trip distance) is $\sim$ 36.7 ms (at the time of this writing, the actual round-trip ping time was 70.69 ms). While read latency from geo-replicas near the location of the client is reduced, the write latency increases substantially. With synchronous writes, this reduces the maximum throughput to 27 writes per second.

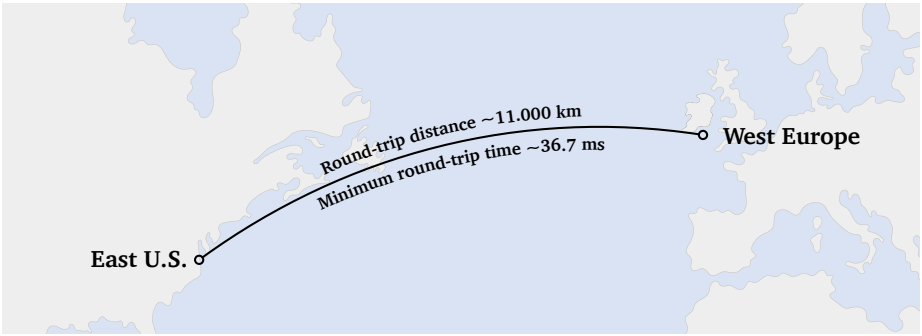


Figure 2.33: Illustration of the round-trip for coordinating between two replicas across continents

For geo-replication, a client-centric consistency model is oftentimes sufficient, as not all data must be available for every client, leading to partial replication. A causal consistency model with partial replication can also be sufficient in this case, as it will only care about causaly related data, but it is difficult to implement [73].

To prepare for disaster recovery, a trade-off between consistency and latency in the non-disaster case needs to be made, and is made on economical risk calculations: what's the cost if stale data is lost forever in the rare case of a disaster? When organizations develop a *business continuity plan*, they need to know the maximum number of lost writes the application can tolerate when recovering from a disaster. The time period of writes that an organization can afford to lose before significant damage occurs is called the *Recovery Point Objective* (RPO). An RPO of 0 means that strong consistency is required across geo-replicas. It should be noted that a small fraction of the system's data will always require strong consistency at a global level, mainly metadata required to manage and monitor the overall state of the system.

Mirroring. The least complex approach, which also ensures that the ordering of operations is preserved, is mirroring of a primary replica into read-only secondary replicas[17]. This can be done with less coordination between the nodes, while it does not guarantee that every write can be read on the secondary replicas immediately: it does not provide strong consistency. During runtime, the secondary replica is updated asynchronously as a *hot backup*, so writes on the primary replica are not blocked. This can happen in a push manner, so that the primary replica send new, committed operations to the secondary replica, or in a pull manner, so that the

[17]Note that the term "replica" here describes a full cluster or even a full availability zone, not just a single intra-cluster node.

secondary replicas fetch the latest operations continously from the primary.

In state machine replication (which is discussed in subsection 2.1.9.2), this can also be done via snapshot installations.

Mirroring is also often used to clone data from a production cluster into a development or testing environment. Extending the distributed architecture to an edge-cloud network, these approaches are also suitable for feeding edge cluster data into one central cloud cluster (which could also be geo-replicated).

2.1.7 Edge Computing

Edge computing (sometimes refered to as *fog computing*) is a paradigm that emerged to cope with the issue that traditional cloud computing is oftentimes no longer sufficient to support the high volume of data processing. In edge computing, operations are executed at the edge of the network, meaning there is an additional layer between the client's device and the cloud on the peripherals of the network that is very close to the client and provides data-locality. It is the consequent next step of geo-distribution. At the edge of the network, services are in general lightweight for local, small-scale data storage and processing [2].

As 5G networks become more widespread, new opportunities for the development of high-volume data processing applications arise, which are naturally suited to edge computing: 5G has the advantages of small delay, large bandwidth and large capacity, which solves many problems encountered in the traditional communication field, but also leads to the rapid growth of data volume. It allows for devices on all layers to share massive volumes of data. Therefore, the development of edge computing technology is closely related to 5G[9].

An edge cloud network comprises the following three layers (as illustrated in figure 2.34 using an event processing example):

- **Terminal Layer**: The terminal layer, also referred to as the *sensor layer*, consists of all types of devices connected to the edge network, including mobile and Internet of Things (IoT) devices (such as sensors, smartphones, smart cars, or cameras). In this layer, the device is not only a data consumer, but also a data provider. Consequently, millions of devices in this layer collect raw data and upload it to the edge layer above, where it is stored and computations are performed.

- **Edge Layer**: The edge layer is the heart of the edge computing architecture. It is located at the periphery of the network and provides computational power

and storage for large volumes of data in close proximity to clients. It stores and computes the data uploaded by the end devices and sends the computation results (usually aggregates of the processed data) to the cloud layer above.

- **Cloud Layer**: The cloud layer integrates all the pre-processed data from the devices on the edge layer. It permanently stores the data and creates global aggregates of all the sub-aggregates that where derived in the edge layer. If the cloud layer should persist all raw data from the terminal layer or just aggregates is up to the requirements of the application provider. The cloud layer is also responsible for computational-intense tasks, such as machine learning applications on data streams fed in from the edge layer.

Consistency Decisions on the Edge. Providing high consistency levels for high-volume data processing applications slows down the whole processing. Even without replication, the round-trip time caused by the physical distance between clients and servers in centralized systems can add significant delay to the response of a request. By physically moving the computation closer to the origin of the data, latency can be significantly reduced. Furthermore, edge-computing allows for a multi-layered consistency model design: the edge appliance can process writes in close proximity to the user with lower network round-trip times and feed this data to the central cloud system, which can be geo-replicated for disaster resilience. This makes the entire system likely to be causally consistent, but at least eventually consistent: writes to the edge appliance can be executed with strong consistency, so that recent successful writes of interest to the current client are immediately readable. Across the edge network, however, it depends on how the data is synchronized with the cloud. While clients near the edge nodes they wrote to will see the writes immediately, all other clients will see them later. In practice, this may not be a problem. It all depends on the design of the application on the different layers: ideally, only derived monotonic aggregates are allowed on the cloud layers, while all non-monotonic operations take place on the edge (see the CALM theorem in subsection 2.1.4.3). In the current literature, there are approaches to apply weaker consistency models to services that previously offered strong consistency, to meet edge-computing requirements [74].

Stream Processing and Complex Event Processing on the Edge. Edge computing naturally comes up when talking about *stream processing* (SP) and *complex event processing* (CEP) [1]. Complex event processing applications are moving to the edge to be able to cope with the increasing volume and throughput of events created by

sensors and mobile devices[75] [76]. The Internet of Things and 5G networks have raised the bar for the capabilities of the underlying event processing architecture, especially for real-time applications. In big data processing tasks, the data records processed are often domain facts and therefore events. Append-only data structures are optimized to work in these environments.

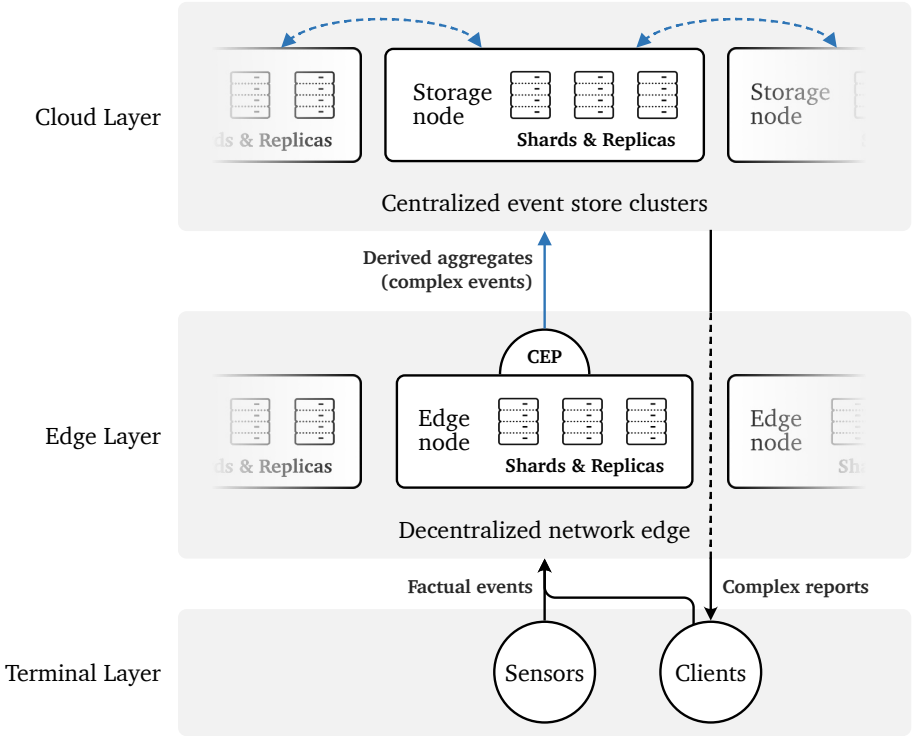


Figure 2.34: Complex event processing in edge-cloud systems

In figure 2.34, the edge appliance derives aggregates (i.e., complex events), and only those aggregates are synchronized with the cloud, where they are further processed and their results can be read by clients again. This provides strong consistency across all layers. This significantly reduces latency, and available computing resources are being used more efficiently compared to a cloud-only deployment of the same application.

2.1.8 Cost of Replication

> The first principle of successful scalability is to batter the consistency mechanisms down to a minimum, move them off the critical path, hide them in a rarely visited corner of the system, and then make it as hard as possible for application developers to get permission to use them.
>
> — James Hamilton, VP & Distinguished Engineer at Amazon

Now that we have discussed the benefits of replication for fault-tolerance, reducing read latency in edge computing and geographically distributed networks, and increasing dependability in general, we need to discuss the downsides: the *cost of replication*. Depending on the degree of fault-tolerance, consistency decisions, and the replication protocol chosen, performance can drop dramatically, especially for write operations. In the following paragraphs, we explain the different dimensions of replication costs, and then briefly discuss appropriate strategies for cost-benefit trade-offs in the following subsection.

Increased Latency and Lower Throughput. A modification on one replica triggers the modification on the other replicas, which must be coordinated depending on the expected consistency level. This messaging and coordination overhead degrades the overall write performance of the system. As shown in the discussion of consistency models in subsection 2.1.4 and described by the PACELC theorem, write latency increases with higher consistency levels. With increasing size of the cluster, the latency is expected to increase at the same time. And as we have shown in subsection 2.1.6, when we replicate across geographic regions with high levels of consistency, we expect even higher latency because we have to cope with the limits of the speed of light.

Availability Trade-Off. As the CAP theorem states, there is a trade-off between consistency and availability. To remain consistent, we must avoid split-brain situations, which reduces overall availability. But in general, availability is remarkably higher compared to running database systems in single-node mode, as we showed in 2.1.3, and in practice even systems with strong consistency can provide five-nine times availability, as we have demonstrated anecdotally.

Hardware Costs. As we have shown in subsection 2.1.2.1, a cluster needs at least $2k + 1$ nodes to reach consensus while tolerating up to k fail-stop failures, while to adress byzantine failures, at least $3k + 1$ nodes are neccessary. That is, to make a

system fault-tolerant, additional hardware of the same class is required, multiplying the cost of acquiring, operating, and maintaining that hardware.

Application Complexity. We have shown that with strong consistency, application developers can expect the distributed database system to behave exactly like a single-node system (see subsection 2.1.4). However, with weaker consistency models, developers must expect to receive stale data and the database system API may look different to provide the hooks and callbacks needed to deal with the asynchronous behavior of such systems.

For developers who implement a replication protocol in their database system, the implementation complexity increases manifold. To be able to coordinate between replicas, all read and write operations must now be designed to be functional, side-effect and context-free, serializable, and atomic, so they can be sent and executed across nodes via messaging. As the complexity of the implementation increases, so does the possibility of causing bugs through faulty code. They must verify that their implementation meets the expected consistency and dependability requirements. Fortunetally, there is the TLA$^+$ (temporal logic of actions) specification language for modeling and verification of such distributed programs. The TLA$^+$ language was introduced by Leslie Lamport in 1999 and comes nowadays with a model checker and a proof system [77]. As an anecdotal example of this, model checking with TLA$^+$ uncovered bugs in the DynamoDB database service on AWS that might never have been uncovered by interactive debugging, as some of these bugs required numerous deeply nested steps of state traces [78].

2.1.8.1 Deciding for Consistency

In order to decide on a consistency model, several factors must be taken into account. The actual consistency model to decide for depends on the use case of the distributed database system. Therefore, many popular DDBS allow the developers to select the consistency model of their choice by configuration. When deciding for a consistency model, it makes sense to base the decision on the dependability properties that are neccessary for the given use case (cf. the bank transfer example in subsection 2.1.2).

There are even attempts to formally describe the consistency model needs at the level of operations: Gotsman et al. propose a proof rule for determining the consistency guarantees for different operations on a replicated database that are sufficient to meet given data integrity invariants [79].

Challenges of Weaker Consistency. When applying a weaker consistency model, especially eventual consistency (cf. section 2.1.9.6), challenges arise on the consuming application side. While a strongly consistent distributed system is similar to a single-node system for the consumer and makes it easy for the developer to use because its API is clear, stale data and long-running asynchronous behavior must be handled appropriately when talking to an eventual consistent system, which makes it a completely different API. Consequently, eventual consistency is not suitable for all use cases.

Multiple Levels of Consistency. The weaker consistency models generally violate crucial correctness properties compared to strong consistency. A compromise is to allow multiple levels of consistency to coexist in the database system, which can be achieved depending on the use case and constraints of the system. It is even possible to provide different consistency models per operation or class of data. An example is the combination of strong and causal consistency for applications with geographically replicated data in distributed data centers [80]. In general, a microservice-oriented architecture that wraps up multiple services into a single application is strongly advised, as each microservice can provide its own specific and well-reasoned consistency model suitable for its particular purpose.

Constantly Review Decisions. When stronger consistency models increase the latency of a system in an unacceptable way and there are no other ways to mitigate this, eventual consistency may be considered. It can dramatically increase the performance of a system, but it must fit the use cases of the system and its applications, and it means additional work for developers. At least, it is worth questioning again and again whether strong consistency is really mandatory: even popular systems like Kubernetes undergo this process, as current research seeks to apply eventual consistency to meet edge-computing requirements [74]. As the authors of the CALM theorem describe, instead of micro-optimizing strong consistency protocols, it is often more effective to question the overall solution design (i.e., perhaps a monotonic solution can be found) and minimize the use of such protocols (cf. subsection 2.1.4.3). At the same time, systems that have originally been eventually consistent could also benefit from re-architecting into stronger consistency to be easier to use and understand for both users and developers.

There are examples that anecdotally show that high availability and high throughput are possible even under strong consistency constraints: AWS S3 introduced

strong consistency for their file operations[18] recently in 2021 [81]. They even claim to deliver strong consistency "without changes to performance or availability", compared to the earlier eventual consistency.

In essence, it is worthwhile to constantly challenge applied consistency models as use cases change or new technologies and opportunities emerge.

Let the User Decide. When deciding on a consistency model for a distributed database system, it is important to recognize that different applications built on top of the database will themselves have different consistency requirements, so it may be a good idea to provide multiple consistency models and flexibility in configuring these models for different types of operations in a database system. Many popular database system vendors allow their users to choose for the consistency model of their choice, even at the operations level. Some of those vendors offer true fine-grained consistency options, such as Microsoft Azure Cosmos DB, which offers even 5 models with gradually decreasing consistency constraints [82].

Immutability Changes Everything. Immutability naturally creates monotonicity, as the set of data—let it be either a payload or commands on this payload, like the `itemInserted/itemRemoved` example in subsection 2.1.4.3 above—can only grow. Helland claims in his paper "Immutability Changes Everything" that "We need immutability to coordinate at a distance and we can afford immutability, as storage gets cheaper" [83]. The latter statement is somewhat reminiscent of Moore's Law.

By designing a system to be append-only, and even better to be monotonic, we lay down the foundation to receive some or all the benefits of coordination-free consistency, depending on the causality of the appended records. Append-only systems not only provide lower latency when replicated, but also better write performance at the local disk level. Many databases are equipped with a *write-ahead log* (WAL) that records all the transaction to be executed to the database in advance. These write-ahead logs allow for reliable and high-speed appends, because records are appended immutably, atomically, and sequentially. The log contains virtually the truth about the entire database and allows to validate past and recent transactions (e.g., in the event of a crash), as well as time travel to previous states of the database, acting like a ledger. Even redo and undo operations are stored in this log. As shown in subsection 2.1.4.3, replicated and distributed file systems depend on immutability to eliminate anomalies. By deriving aggregates from append-only

[18] Bucket configuration operations are still eventual consistent in AWS S3 at the time of this writing.

logs of observed facts, consistency can be guaranteed to a certain degree[19]. From a particular perspective, a database is nothing more than such a large, derivative aggregate. Append-only structures also increase the safety of a system and thus support its fault-tolerance: if a system is limited to functional computations on immutable facts, operations become idempotent. Then the system does not become faulty due to failure and restart.

When building a distributed system and thinking about the consistency models, it is therefore useful to think about the nature of the data that this system will store and manage in advance. Table 2.8 helps in categorizing data into *inside data* and *outside data*, as described by Helland [83]. The latter is immutable and therefore allows for coordination-free consistency under certain circumstances.

Table 2.8: Categorization of inside vs outside data

	Inside Data	**Outside Data**
Changeable	Yes	No: Immutable
Granularity	Relational field	Document, file, message or event
Representation	Typically relational	Typically semi-structured
Schema	Prescriptive	Descriptive
Identity	No: Data by values	Yes: URL, document id, UUID...
Versioning	No: Data by values	Versions may augment identity

Influence of the Network Infrastructure and Overall Architecture. Not only the use case, but also the capabilities of the infrastructure the system will be deployed onto (i.e., the network) and the overall technical architecture play an important factor in the decision. Under certain circumstances, it is possible to provide high levels of consistency and yet low latency and availability, e.g., by using multiple or even nested layers of different consistency models and intelligent partitioning techniques.

The critique of the CAP theorem presented in the previous subsections allows for a more deliberate choice of consistency in practical systems, since several other properties can affect the actual requirements for consistency and dependability, often even more than the original theoretical properties of the CAP theorem. As

[19] In large scale distributed systems, the consistency of such logs can still be victim to tampering and other security threats, as it is the case for blockchains (cf. subsection 2.1.10.4).

an example, Google's distributed *NewSQL* database system Spanner is in theory a CP class system [84]. It's design supports strong consistency with real-time clocks. In practice, however, things are different: given that the database is proprietary and runs on Google's own infrastructure, Google is in full control of every aspect of the system (including the clocks). The company can employ additional proactive strategies to mitigate network issues (such as predictive maintenance) and to reduce latency in Google's extremely widespread data center architecture. In addition, intelligent sharding techniques have been deployed (we discussed partitioning and sharding in subsection 2.1.5) that take advantage of this data center architecture. As a result, the system is highly available in practice (records to date even show availability of more than five nines (99.999 %) at the time of writing), and manifested network partitions are extremely rare. Eric Brewer, the author of the CAP theorem and now (at the time of writing) VP of infrastracture at Google, even claims that Spanner is technically CP but effectively CA [85]. It is important to realize that this is difficult in practice for open, self-managed distributed databases, or generally for smaller, less complex infrastructures, or when there is no control over the underlying network, as this requires a joint design of distributed algorithms and new network functions and protocols (cf. the Eris protocol in subsection 2.1.9.7). And after all, it is always a question of overall economic efficiency.

In addition, the actual choice of a replication protocol adds another layer of considerations that must be factored into the decision. As a rule of thumb, the more tightly coupled the replicated database system and infrastructure, the easier it is to ensure strong consistency without compromising latency and availability.

2.1.8.2 Further Cost Reduction Strategies

Service Decoupling. The additional cost of hardware can be mitigated by separating compute-intensive application services from the data store, as shown in figure 2.35. As storage becomes cheaper (and, in the long term, compared to the costs of unavailability and failures, providing *zero marginal cost*), the data store can be deployed on multiple less powerful nodes with high storage capacities, while the application services are deployed on more powerful machines but with lower storage capacities and a much lower replication factor. If the services are designed to be stateless, replication of the services is only required for availability and load balancing, since these services do not need to be coordinated with each other.

Partial Replication. Not all data must be replicated, and not all with the same level of consistency. With partial replication, each replica holds only a subset of

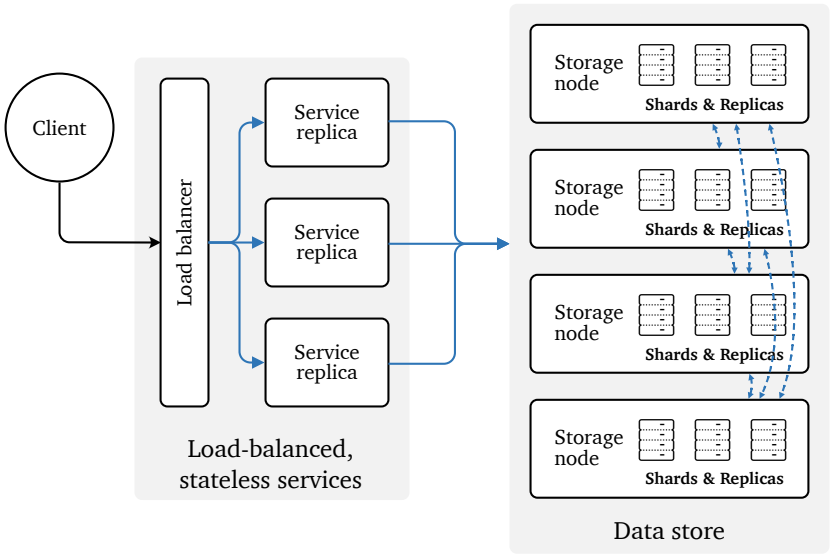


Figure 2.35: Service replication and data replication are different: while compute-intensive, stateless services can be deployed on powerful machines with a lower replication factor for load-balancing and high availability, data is replicated on more, but less powerful nodes for fault-tolerance and dependability.

all data. This is often used for geo-replication: the data is replicated to nodes in close proximity to the respective clients. The replica subsets are distributed over multiple nodes and clusters in a way that in case of a node or cluster failure, the total set of data can be recovered from the various partial replicas. This is shown schematically in figure 2.36. This can be achieved with sharding where the shard keys also correlate with data locality.

Elastic Horizontal Scaling. With *elastic scaling* (often referred to as *auto-scaling*), it is possible to maintain a consistently high throughput rate even as requests to the system increase. Elasticity means that new shard nodes are automatically provisioned on additional hardware resources and the load is balanced if a trend towards higher throughput in the long term is detected (and vice versa, if the trend is downwards, then superfluous resources are removed again). This requires constant monitoring of the system. With a lot of clients served this way, the costs for replication converge to marginal costs compared with the increased throughput and constant latency provided to clients. In append-only systems, this gets more complicated as sharding a single event stream means that it must be merged again on queries.

The impact of horizontal scaling on the overall system performance can be more

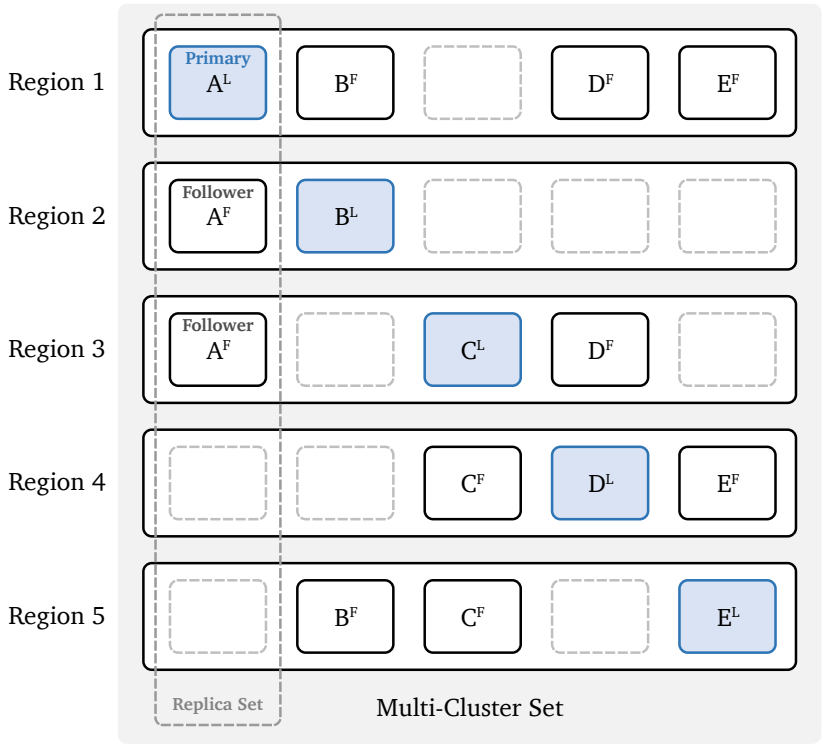


Figure 2.36: Example scheme for partial replication. The same pattern can be applied intra-cluster (by load-balanced partial replication of shards) and inter-cluster (by data-locality-aware geo-replication of shards or even whole clusters).

significant than the choice of consistency model: even with eventual consistency, the system will never become consistent if the average write rate remains higher than the maximum throughput of the current replica shards, since the replica states will never converge even if the current primary node accepts such high write rates.

Smart Engineering. Since the network overhead can quickly become the most expensive part of an operation in a data store, it makes sense to send and execute operations in batches to reduce the share of the network overhead on the total processing time. The batch can be processed atomically, much like a transaction: either all or none of the operations in the batch are executed; however, it can also be executed optimistically: executing as many operations of the batch as possible. Both approaches come with a new cost: the risk of data loss if the batch is not completely written. If the node processing the batch fails over, it must compare what has already been written and what have been in the batch (which should have been logged in a

write-ahead-log) if we want to provide clients with fire-and-forget behavior.

The batch behavior can be enforced by using a *buffer* on the processing node. The buffer can not only reduce the overall network overhead, but also temporarily cushion unusually high write rates if the average rate remains below the maximum throughput (if the average rate stays over the maximum throughput rate for a longer period of time, the overall system will slow down and likely crash).

As we have shown in subsection 2.1.8.1, it makes sense to think beyond the application layer and to start questioning the network layer, in case you are in control of the underlying network architecture and protocols. Under certain circumstances, latency and availability can be kept high even with a high level of consistency, and even with transaction management, as we will show in subsection 2.1.9.7.

Note that it is worth questioning the consistency model or the implementation itself before considering smart but complex engineering techniques to improve latency and availability while maintaining a strong consistency model (see subsection 2.1.8.1).

2.1.9 Theoretical Replication Protocols

Now that we have discussed dependability, consistency models, and the costs and trade-offs of replication, we can finally get more specific: the next subsections describe the different categories of replication protocols and follow with a discussion of relevant protocols.

2.1.9.1 Consensus Protocols

We start looking into replication protocols that provide strong consistency, namely *consensus protocols*. The first research on consensus protocols was about concurrent memory access by multiple threads or processes on the same machine. The same principles apply to nodes in a distributed system writing to distributed memory or persistent storage, since the main issue is the agreement on a single value to be written. The first mention of a consensus algorithm in the distributed system literature was 1979 in a paper by Robert H. Thomas [86].

Consensus protocols have been described to solve the *consensus problem* [87]. The consensus problem describes the problem that a majority of nodes in a distributed system must agree on a single value. Consensus protocols describe how this majority can be achieved by specifying the necessary communication steps for coordinating between nodes, and between nodes and the client. All this happens in the context of fault-tolerance: nodes must agree on a common value even in the presence

of failures, while providing a high level of availability without compromising on consistency. This can be summed up as follows: "a consensus protocol enables a system of n asynchronous processes, some of which are faulty, to reach agreement [31]."

Consensus protocols typically have the following properties [88]:

- They never return a wrong result under all faulty conditions (either fail-stop or even byzantine, depending on the protocol), including network partitioning, packet loss, duplication and reordering.
- A system is available as long as the majority of the nodes are operational and can communicate with each other and with the clients.
- They do not depend on external timers to ensure the consistency of the logs.
- Under normal conditions, a write request can be served as soon as an absolute majority of the cluster has agreed on it; a minority of slow servers will not affect the overall performance of the system.
- Each process starts with some initial value; at the conclusion of the protocol all operational nodes must agree on the same value.

Following are some example applications where consensus is needed:

- Clock synchronisation,
- Google PageRank,
- smart power grids,
- cluster metadata management,
- service load-balancing,
- container orchestration (such as Kubernetes),
- key-value stores in general,
- distributed ledgers and ultimatively the Blockchain (note that some of the consensus protocols here implement less strict consistency models, but are still called consensus protocols).

Synchronous vs. Asynchronous Consensus or: the FLP impossiblity result. Certain different variations of how to understand a consensus protocol appear in the literature. They differ in the assumed properties of the messaging system, in the type of errors allowed to the processes, and in the notion of what a solution is. Most notable is the differentiation between *consensus protocols for asynchronous and synchronous message-passing systems*. Fischer et al. have proven in the famous *FLP impossibility result* (called after their authors) that a deterministic consensus

algorithm for achieving consensus in a fully asynchronous distributed system is impossible if even a single node crashes in a fail-stop manner [89]. They investigated deterministic protocols that always terminate within a finite number of steps and showed that "every protocol for this problem has the possibility of nontermination, even with only one faulty process." This is based on the failure detection problem discussed earlier in subsection 2.1.2.1: in such a system, a crashed process cannot be distinguished from a very slow one. The FLP result is an important finding for theoretical considerations.

As pessimistic as this may sound, the FLP result can easily be mitigated in practice in one of two ways:

- Assume synchronicity in the protocol,
- Add nondeterminism to the protocol.

Bracha et al. consider protocols that may never terminate, "but this would occur with probability 0, and the expected termination time is finite. Therefore, they terminate within finite time with probability 1 under certain assumptions on the behavior of the system[20]" [31]. This can be achieved by adding characteristics of randomness to the protocol, such as random retry timeouts for failed messages, or by the application of *unreliable failure detectors* as in the *Chandra–Toueg consensus algorithm* [34]. Additionally, adding pseudo-synchronous behavior like in Raft (which is described in detail in section 2.2), where all the messaging goes in rounds in a bounded timeframe by enforcing a message enumeration and adding an upper bound for message time by using timeouts, removes the initial problem of asynchronous consensus.

Centralized vs. Decentralized Consensus. In *centralized consensus*, an additional authority is responsible for coordinating the nodes when a majority vote is pending. The *Zookeeper Atomic Broadcast* (ZAB) protocol is one example for such a centralized protocol [90]. Such centralized protocols are problematic as the additional coordination service is the bottleneck—if the service can not be reached due to a failure or network partitioning, the whole system becomes unavailable—and must therefore be replicated with strong consistency, too.

In *distributed consensus*, a self-managed quorum is responsible for coordination. Such protocols do not rely on an additional service because the protocol is built in the replicated system itself. There are protocols with strong single-leader characteristics,

[20] So, to be formally correct, we say that nodes in a consensus protocol *eventually* agree on a value and

where only a single node of the quorum is allowed to serve both read and write requests, and there are leader-less protocols, as well as combinations of both. Popular examples for distributed consensus protocols are *Paxos* (subsection 2.1.10.1) and Raft (section 2.2), the latter being the focus of this work.

Leader-less *decentralized consensus* is the only option for consensus in highly decentralized multi-agent systems, such as *blockchains*. Decentralized consensus enables many actors to persist and share information securely and consistently without relying on a central authority or trusting other participants in the network.

Single-Value vs. Multi-Value Consensus. Another way to categorize consensus protocols is by the set of values to which the protocol refers:

In *single-value consensus protocols*, nodes agree on a single value. Those protocols are not designed to agree on a consistent ordering of operations. Originally, the Paxos (see subsection 2.1.10.1) algorithm was described as a single-value consensus protocol. In the literature, the word consensus refers to single-value consensus in general.

In *multi-value consensus protocols*, the nodes also agree on a consistent ordering of the operations and values, forming a progressively-growing operation history. The correctness requirement is thereby two-fold: correct execution results for all requests and correct order of these requests. This may be achieved naively by running multiple iterations of a single-value consensus protocol that has been extended to include timestamps, but many optimizations of coordination and other considerations such as reconfiguration support can make multi-valued consensus protocols more efficient in practice. *Multi-Paxos* is an example for such a protocol.

Crash-Fault-Tolerant vs. Byzantine-Fault-Tolerant Consensus. Yet one more way to categorize consensus protocols is by fault-tolerance. As described in subsection 2.1.2, a consensus protocol is called k-fault tolerant if in the presence of up to k faulty nodes it reaches consensus with probability 1.

A protocol that solves the consensus problem in the face of faulty nodes must guarantees the following properties (extending the liveness and safety properties as described in subsection 2.1.2):

- **Termination**: Every non-faulty node eventually decides on some value,
- **Integrity**: If all non-faulty nodes proposed the same value y, then any non-faulty node must decide y,

an order.

- **Validity**: The agreed-upon value must be the same as the initially proposed value,
- **Agreement**: Every non-faulty node must agree on the same value.

Crash-Fault Tolerant (CFT) Consensus Protocols. A *crash-fault tolerant* (CFT) consensus protocol provides fault-tolerance for fail-stop failures, but not for byzantine failures. We have shown in subsection 2.1.2.1 that for a system to tolerate up to k faults, at least $k+1$ nodes are required, but this only accounts for a read operation perspective. For such a system to still be able to reach consensus on write operations, even more nodes are required: $\lceil (n+1)/2 \rceil$ correct nodes, thus more than half of all nodes, are necessary and sufficient to reach agreement [31]. We call this number a *quorum*. Based on this number, we know how big a cluster must be to tolerate k crashed nodes: $n \geq 2k+1$ where n the size of the cluster.

A consensus protocol is fault-tolerant by masking failures. But in practice, they also apply failure detection: to detect a crash-failure, nodes in a quorum typically send periodic heartbeats. If for some node no heartbeat was received for a certain timeout period, the protocol assumes that this node is dead and removes it from the quorum. In the literature, there are more sophisticated approaches to failure detection, but for distributed databases, heartbeats are most common in practice and easy to implement.

Byzantine-Fault Tolerant (BFT) Consensus Protocols.

> It is not sufficient that everyone knows X. We also need everyone to know that everyone knows X, and that everyone knows that everyone knows that everyone knows X — which, as in the Byzantine Generals' problem, is the classic hard problem of distributed data processing.
>
> — James A. Donald

A *byzantine-fault tolerant* (BFT) consensus protocol provides fault-tolerance for byzantine failures. The problem of byzantine faults and byzantine consensus was first described by Leslie Lamport in the *Byzantine Generals Problem* [91]. It is a classic problem in distributed systems that is not as easy to implement, adapt, and understand as it might seem to a system architect.

The Byzantine Generals Problem describes a situation where a number of generals besiege a town with their armies, surrounding this town to plan a concerted attack. They can only win this attack if they all attack together at once, so they need to agree on the time of the attack. They can also decide together to retreat. They also

assume that some of the generals are disloyal and will send out false commands. Due to the distance between the generals, they can only communicate with each other via messengers on horseback, so there is no way of verifying the authenticity of such a message. Lamport et al. have shown that the problem can only have a solution if more than two-thirds of the generals are loyal. Figure 2.37 illustrates the byzantine problem for three nodes, which can not be solved.

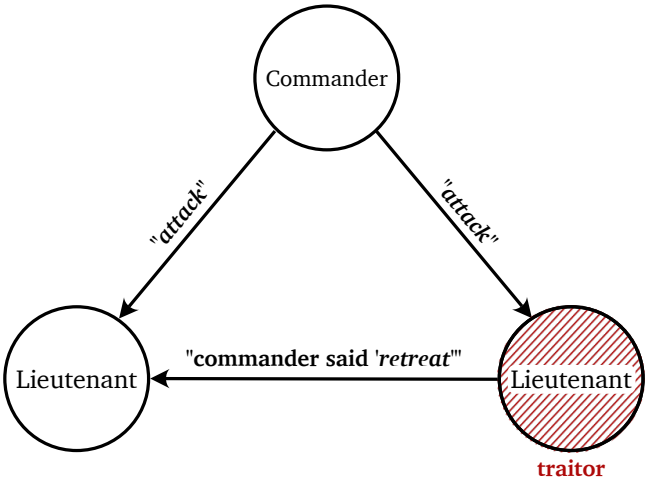


Figure 2.37: The byzantine generals problem can not be solved with only three nodes

For read requests, a cluster must have at least $2k+1$ nodes so a majority of nodes can still respond with the non-arbitrary values when k nodes are faulty, as shown in subsection 2.1.2.1. But to be byzantine-fault tolerant BFT for write operations, we need more nodes to be able to agree on the correct value. The quorum must be significantly larger than that for crash-fault tolerance: for any consensus protocol to be byzantine-fault tolerant, a quorum of $\lceil (2n+1)/3 \rceil$ correct nodes are necessary and sufficient to reach agreement. Therefore, to tolerate up to k faulty nodes, a cluster must have at least n nodes with $n \geq 3k+1$. This fundamental result was first proved by Pease, Lamport et al. [92] and later adapted to the BFT consensus framework [31].

Every byzantine-fault tolerant consensus protocol is also fail-stop-fault tolerant [93]). Blockchain protocols generally belong to the class of BFT consensus algorithms, as crashed nodes are not a problem in general, but faulty values are. We describe this briefly in subsection 2.1.10.4.

We won't go into much details here, as this work is limited to a problem scope where byzantine faults are rare (but not yet excluded).

Weighted quorums. Some consensus algorithms, like the Proof-of-Stake algorithm used for certain blockchains and cryptocurrency tokens, allow for weighted quorums. As a rule in general consensus protocols, the votes of all quorum members have the same weight in a vote. With weighted quorums, however, there may be cluster members whose vote is more important. Whether a system is k-fault-tolerant or not is now judged by the sum of the relative weights rather than the number of nodes. To perform a write operation, an agreement of the majority of the nodes is now no longer mandatory, if among them there are nodes whose vote has relatively more weight.

Network reconfigurations. If a node is removed from the cluster because it is faulty or has not responded for a certain period of time, and is then restored again, it attempts to rejoin the cluster. Also, when the system is scaled horizontally by adding new nodes, these nodes can join the cluster if the load balancer assigns them to do so, i.e., for a new shard. In the case of network partitioning, many nodes can no longer be reached, but the system should remain available. Or due to planned maintenance, the hardware of the nodes is upgraded, or the IP addresses of the nodes change if not statically assigned. In the case of blockchains, new nodes are added and removed at very short intervals. In all of these cases, the network is reconfigured, and generally clients are not expected to notice, i.e., the system remains available and consistent while the network is reconfigured, and latency does not degrade. The consensus protocols used in practice generally enable such seamless network reconfigurations, especially the decentralized protocols.

2.1.9.2 State Machine Replication

The *state machine replication* (SMR) protocol describes a protocol with strong consistency that is based on the simple idea of a *deterministic state machine* and a set of operations[21] that can be executed on it. Every node of the distributed system represents such a state machine, and operations can be requested by clients of the distributed system like they would request them on a non-replicated state machine on a single server. A central idea of SMR is a serializable write-ahead log where all requested operations are written to in the order of their proposed execution before the actual execution. This log is replicated between all nodes of such a cluster, therefore the core protocol of SMR is referred to as *log replication*. An entry is only written to the log after a majority of nodes agreed on it. These operations will

[21] We will use the terms *operation* and *command* interchangeably throughout this work.

be executed on the state by the individual state machines in the order that they appear in that log. All these operations are functional and side-effect free, and since the state machine is deterministic, all the servers will produce the same sequences of states, resulting in the states of all nodes always being consistent. This allows for an reduction of the state of a distributed system and its individual nodes to an append-only log. The SMR approach is illustrated in figure 2.38.

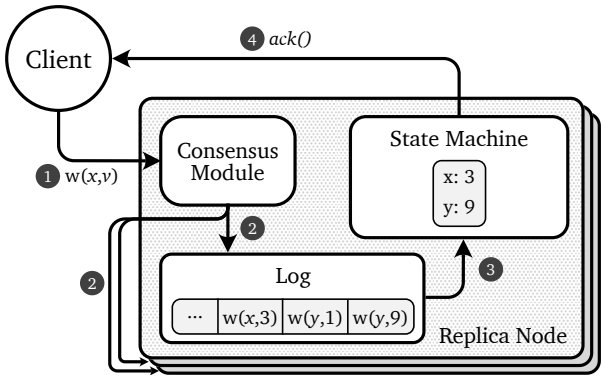


Figure 2.38: Illustration of the messaging between clients and cluster in the state machine replication approach

State machine replication is the protocol of choice in many popular services and the subject of numerous academic studies, originating in Leslie Lamport's early work on clock synchronization in distributed systems [53]. Lamport shows that "a distributed system can be described as a particular sequential state machine that is implemented with a network of processors. The ability to totally order the input requests leads immediately to an algorithm to implement an arbitrary state machine by a network of processors, and hence to implement any distributed system". In the original paper, the SMR approach wasn't discussed in the face of failures: "The problem of failure is a difficult one, and it is beyond the scope of this paper to discuss it in any detail." The first formal description of the SMR approach to provide fault-tolerance was presented by Schneider [94] in 1990.

In the previous subsection, we investigated the consensus problem, consensus protocols and their various categories. In past years, the relationship between SMR and consensus has been the subject of numerous studies. Consensus is the basis for state machine replication and its main component. State machine replication introduces the perspective of a consistent state. Log replication—its core component—in turn extends decentralized multi-value consensus protocols by introducing the perspective of a consistent log. This consistent log and state result from consensus on

both operations and the order of these operations to execute: all nodes must agree on the same state, even if some nodes crash or disconnect, which requires consensus for each command, but also to execute these commands in the same order, otherwise different replicas may end up in a different final state. We could say that SMR is an approach to solving the consensus problem from an engineering perspective, while consensus protocols give a theoretical perspective on this problem [87].

The state machine can be formally described by [53]

- a set C of possible commands,
- a set S of possible states,
- and a state transition function $\delta : C \times S \rightarrow S$ with $delta(c,s) = s'$, $c \in C$ and $s, s' \in S$.

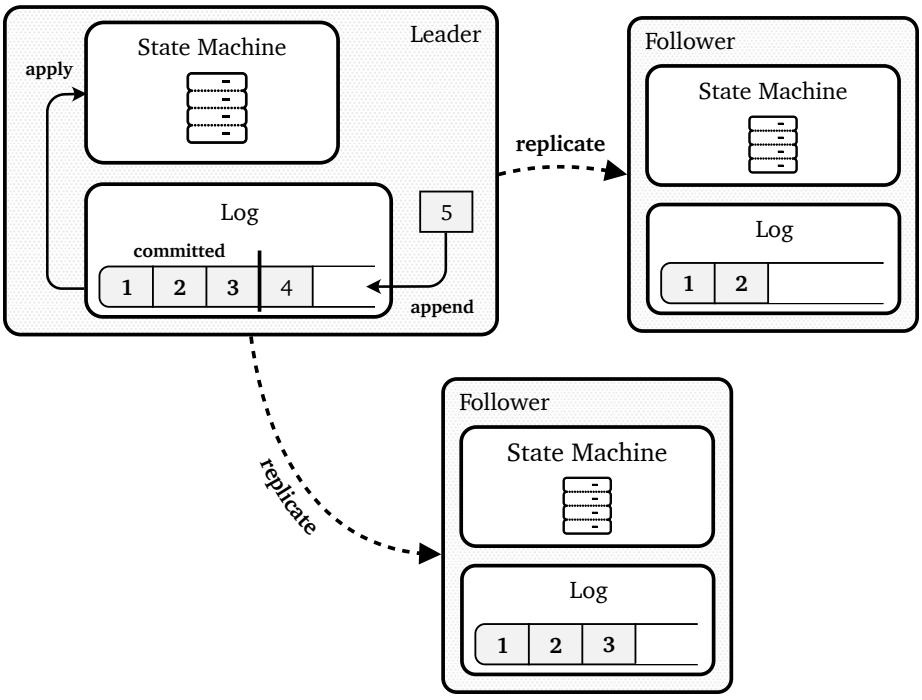


Figure 2.39: Illustration of the log replication mechanism in the state machine replication approach

The SMR approach sits in the client-server realm, where it tries to emulate a single machine towards a set of clients. We can extend safety and liveness in this sense from the point of view of the log:

- **Safety**: All nodes store the same prefix in their logs, i.e., if a node appends a at the index i to its log, then no other node will append a different command $b \neq a$ at this index i to its log.
- **Liveness**: All nodes will eventually apply a command issued by a client, i.e., if a client issues a command c, then eventually all nodes will have a log entry c appended at some index j in the log and all previous indexes $i < j$ in the log will be set.

For safety, this requires the following guarantees [93]:

- All server nodes start with the same initial state,
- *Total-order broadcast*: all nodes receive the same sequence of requests in the order they were issued by clients,
- All nodes receiving the same request will always output the same execution result and end up in the same state.

Single Leader Election. In state machine replication, there is commonly a concept of a single, strong leader. A strong leader node is the only instance in a SMR protocol that is allowed to communicate with the client and to coordinate writes—and often also reads—between replicas. This ensures linearizability, as the single leader can ensure that commands will be ordered in the same order as a client has requested. The leader must be somehow decided on between the replicas, and this must happen with strong consistency guarantees. This process is called *leader election*. In the same way as consensus and SMR agree on a single value or a order of commands, they must agree on a leader decision, i.e. through another instance of consensus. The FLP impossibility result (as described in subsection 2.1.9.1) hence implies that a reliable leader election protocol must use either randomness or real time—for example, by using timeouts.

The Generation Clock. State machine replication mitigates the FLP impossibility problem by adding pseudo-synchronous behavior through timeouts and a *generation clock*. If no progress is made in deciding on the next command in a certain amount of time, or when a new leader is elected due to a leader failure or timeout, a new round of the protocol begins, in which the steps may be repeated again. This rounds are marked by a generation clock, sometimes refered to as a *term* or *epoch*. The generation clock is allowed to only increase monotonically. The clock ensures that in case of failures and network partitions, once a node rejoins a quorum,

messages issued by that node (i.e. a leader node) that are associated with an outdated generation clock are ignored, and the node can synchronize itself again with the new clock. In case of network partitioning, meanwhile committed entries of the partitioned nodes may be dropped to restore consistency across the whole distributed system. The generation clock is persisted together with the log entries, so replica nodes can use this information to find conflicting entries in their log after a fail over.

The High-Water Mark. In a system with no failures, the commands could be discarded after being applied to the state machine. But in the face of failures, the system must know what has been successfully committed and what is still pending, to restore replica states from the log[ˆlogless]. For strongly consistent replies to read requests, any cluster node is only allowed to return a state based on the latest log entry that was successfully replicated to a quorum. The *high-water mark* is an index into the log file that marks this latest log entry. During instances of the replication protocol, the high-water mark is passed to replicas. All servers in the cluster should only allow reads to clients that reflect updates that are below the high-water mark.

[ˆlogless] There are approaches to consistent, logless SMR protocols that still provide safety and liveness, such as proposed by Skrzypzcak et al. [95]: instead of agreeing on a sequence of commands to build up a consistent log, nodes in logless SMR agree directly on the sequence of state machine states.

As we have pointed out in subsection 2.1.8.1, “immutability changes everything”. Append-only data structures make it easier to achieve and maintain consistency. The state calculated by the state machine is a large derivative aggregate. The current state can always be reconstructed by applying the committed portion of the log up to the high-water mark:

$$\delta(log_{\text{hwm}}, s) = \delta_{cmd_i} \circ \cdots \circ \delta_{cmd_0}(s)$$

It is theoretically possible to replay only a part of the log to achieve a time-travel query, similar to event sourcing. In practice, the command log grows unbounded and depending on the state that is managed, such time-travel queries are too expensive, yet there are event sourcing systems built on replicated state machines where the state is an optimized append-only data structure holding the events of interest for sourcing, while the command log contains more events and operations (such as node management and other metadata commands that are not actual payload). The unbounded nature of the log is mitigated in different ways: one way is to keep the log transient in-memory instead of persisting it on disk. This makes fail over more

complicated, but it is possible under circumstances if some features such as network reconfiguration support are not required.

Snapshotting and State Transfer. Another way to keep the log size bounded is to apply *log compaction*. *Snapshots* of the current state are created periodically or as a reaction to certain events or commands in the log. These snapshots also hold an index to the latest log entry that was included to derive this state. All log entries to this point are then removed from the log. We illustrate this in figure 2.40. When a new node joins the cluster, instead of all the log entries, the latest snapshot and all additional committed log entries since then are sent to this new node.

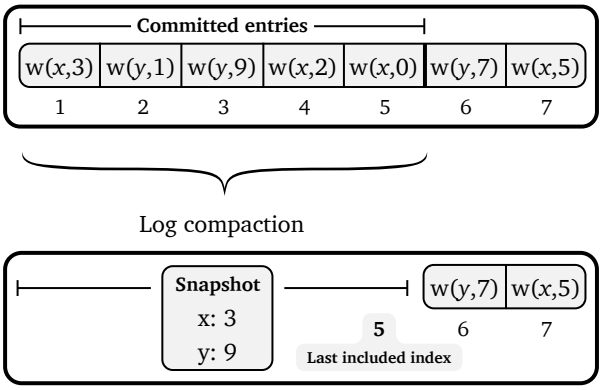


Figure 2.40: Compaction of committed log entries by creation of a state snapshot

Log Truncation. When a server joins the cluster after crash/restart, there is always a possibility of having some conflicting entries in its log. So whenever a server joins the cluster, it checks with the leader of the cluster to know which entries in the log are potentially conflicting. It then truncates the log to the point where entries match with the leader, and then updates the log with the subsequent entries to ensure its log matches the rest of the cluster.

This can be conflicting with log compaction. If the latest log entry that happens to be valid after truncation is already compacted, the whole state must be transfered, which is more expensive then trasmitting only a diff of the log entries.

Network Reconfiguration. Network reconfiguration support in SMR is evenly important to those in Consensus, see subsection 2.1.9.1. In addition, in SMR, a network reconfiguration also includes a state transfer, where a newly joined or rejoined node receives and installs the latest state snapshot.

Failure Detection. In SMR, failures are generally masked. But to provide a single leader, a failure detection mechanism is neccessary to detect a leader failure, to prevent the system from being unavailable during leader crashes. In most SMR protocols, this is realized with heartbeats and a timeout. Once a node does not receive a heartbeat from the leader anymore in a certain timeframe, it assumes that the leader crashed and starts a new leader election, which also triggers an increment of the generation clock.

Relation to the Consensus Problem. State machine replication uses consensus under the hood to agree on a single value in a round of voting. But additionally to consensus on a single value, it adds other properties to the protocol, especially the ordering mechanism. Although there is also a consistent ordering in multi-value consensus, the state machine approach is different: it approaches the problem from the perspective of the state, not the consensus. Or, as Antoniadis et al. note: "Solving consensus is one step away from implementing state machine replication" [96]. They have shown that SMR is more expensive than consensus from a complexity point of view, and even under synchrony conditions, no SMR algorithm can guarantee bounded response times compared to Consensus.

Relation to Weaker Consistency Models .

It appears that the state can be seen as a derived non-monotonic aggregate, while the commands in the log are trivial facts and operations on it. From this point of view, if the state appears to be or can be designed as a monotonic aggregate, weaker consistent replication protocols are more suitable, such as optimistic replication (see subsections 2.1.4.3 and 2.1.9.6).

Byzantine-Fault Tolerance .

All these safety and liveness properties described in this subsection do not apply to byzantine failures, i.e., they only apply to *honest* nodes. Basic state machine replication is only crash-fault tolerant, while there are also byzantine-fault tolerant state machine replication approaches, such as *Practical Byzantine Fault Tolerance* (PBFT) [97]. Since byzantine-fault tolerance is not the focus of this paper, we will not discuss it in detail here, but refer to the relevant literature [98].

2.1.9.3 Read One / Write All (ROWA)

Read one / write all is the simplest, yet weakest approach to strongly consistent replication. Its two properties are:

- **Read one**: Any replica node can answer a read request, even if all other nodes failed.
- **Write all**: All replicas can be written to from any client, and the write is only confirmed successfully if it is replicated on all replica nodes.

In ROWA, the cluster is immediately unavailable for writes once a single node crashes. In addition, the write latency grows with each replica in the cluster, as every node participates in writing and must acknowledge their writes. The latency is therefore at least as high as the longest latency of any node, rendering slow nodes as a bottleneck. In quorum-based replication, as we presented in consensus and state machine replication, a single unresponsive or slow node will never be a bottleneck if a quorum of responsive nodes can be achieved.

In contrast, for read operations, the availability is very high and the latency low since a data object can be retrieved from any replica without delay.

2.1.9.4 Primary-Copy Replication

When we allow for transactions instead of only single values, replication with ROWA becomes more unstable and prone to problems: as the workload scales up, it becomes exponentially slower and prone to deadlocks or reconciliations, since all nodes must coordinate with each other pair-whise. The *primary-copy replication* approach was introduced in 1996 to reduce this problem [99]. There are a variety of primary copy replication approaches, ranging from simple and weak approaches that make some trade-offs in fault tolerance, such as continuous backups of relational databases, to strongly consistent approaches that include state machines and atomic transfers.

In a primary-copy replication setup, there is one node holding a primary-copy of the data, while other nodes provide read-only backups of the data. Unlike state machine replication, primary-copy replication does not replicate the commands themselves, but the results of their execution—which is the diff of the state before and after the execution [100]. This also means that in all cases commands are first executed on a primary copy of the data before their results are replicated. The disadvantage of this approach is that this does not ensure strong consistency, but only sequential consistency of distributed copies: what is written, can be read immediately only on the node holding the primary copy, while it appears on the replicas with a delay, but at least in the same order. This is a reason why there are applications of primary-copy that prevent reading from the replicas and only use them as backups for fault-tolerance, which re-introduces strong consistency as long as the primary copy does not fail (when this happens, rollback strategies must be in

place, which is not always possible and error-prone). This is done, for example, in the *ZooKeeper Atomic Broadcast* protocol (see subsection 2.1.10.2 for details).

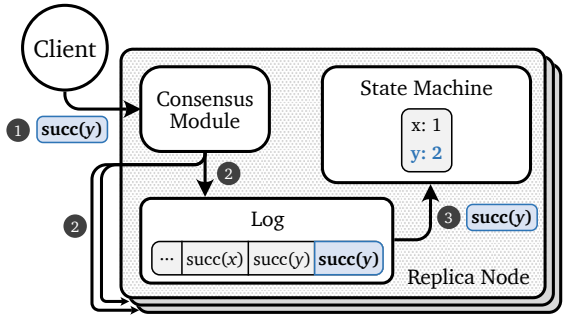
Compared to ROWA, this system can tolerate failures, at least those of the backups. But, primary-copy nodes can easily become a bottleneck in the system. It does not scale with the number of nodes, similar to state machine replication and consensus. What happens if the node serving a primary-copy fails? Failure detection mechanisms are required to detect such situations and subsequently select an alternate primary-copy server (similar to the leader election problem of state machine replication), creating the risk of multiple primary-copies when different network partitions occur; a problem that is mitigated in some protocols. In addition, a failure of a primary-copy can cause loss of data not yet replicated. Until the instantiation of the backup, the system may also unaivable to the user.

Primary-copy is often used in RDBMS to provide continously updated hot backups. There are different approaches in primary-copy replication to handle transactions. One approach involves performing transactions atomically on the primary copy first, before passing the result of the transaction to the replicas.

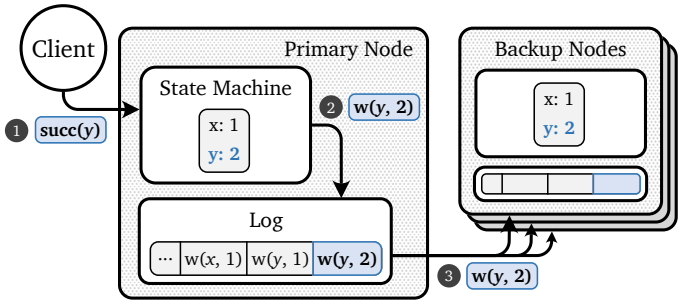
Primary-Copy with Logs and State Machines. It is possible to implement the primary-copy approach with a replicated log and state machines. Instead of the actual commands, this log holds the outcomes of the command executions, but still in a strict sequential ordering. The replication is achieved after the execution of a client request by the primary copy by writing the state diff to a replicated log, which can happen in a similar same way as in state machine replication by applying consensus respectively total order broadcasts. This is illustrated in figure 2.41. To achieve linearizability for strong consistency, the effects of writes should not be made visible to clients until the log entries have been committed to a quorum, which is far from trivial to implement. In addition, the primary copy should also contain the clients' responses in the log entries so that the backup servers can return the same response if the clients retry.

2.1.9.5 Active Replication

In *active replication*, the clients are responsible to ensure consistency, therefore they participate actively in the protocol. Once a client issues a request, it must send this requests to all the replicas it wants the data to be replicated to, in contrast to *passive replication*, where only a single node receives the request and then takes care of the replication (such as in the other protocols shown so far). All replicas are equivalent, there is no leader node. The replicas take the requests in sequential



a) Replicated state machine approach



b) Primary copy approach

Figure 2.41: State machine replication vs primary-copy replication. a) In state machine replication, the replicated log contains the actual commands and are consistently replicated to the logs before they are executed on the state machines. b) In primary-copy replication with state machines, the primary copy executes the command before the replication starts. After execution, it replicates the outcome of the execution, which is the diff between the previous and the new state, to the replicated logs, from where it is applied to the state machines.

order and execute them. After the execution, each individual node replies to the client. There is no direct communication between replicas, as all the coordination happens through the client. The client decides if it accepts the result of the write (i.e. one node, a quorum or all nodes acknowledged it), which also allows the client to compare the results to cope with byzantine errors (up to $n/2-1$, see subsection 2.1.2.1 for reference). All dependability, consistency and performance properties therefore depend on how the clients act, e.g., if they wait for all nodes to acknowledge a write, the slowest replica is the bottleneck.

This approach does not provide strong consistency, as the replicas immediately execute a command once they receive it, and this can happen at different points in time, depending on the individual latency and load of the nodes. Moreover, multiple

clients may invoke the system simultaneously, and even if they do not write at the same time, the order of their requests to each node can only be guaranteed to follow the program order, ensuring at least sequential consistency (cf. the sequencing in figure 2.15 of subsection 2.1.4). Unless a client waits for all nodes to acknowledge a write, sequential consistency is also not guaranteed, as some intermediate writes may never have been performed by some nodes.

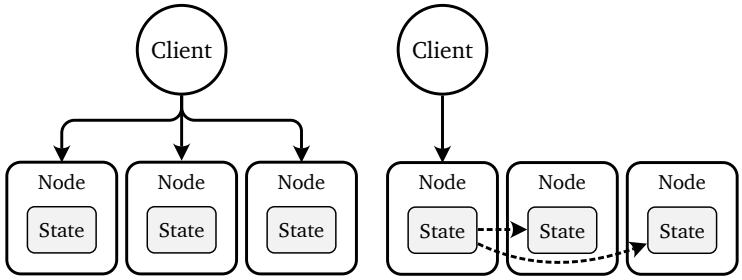


Figure 2.42: Active vs. passive replication

2.1.9.6 Optimistic Replication

There are many use cases where the cost of replication is too high when synchronization between nodes in a replica set is required, especially for real-time live collaboration applications or applications that cannot be guaranteed to be online all the time (such as mobile apps). In these cases, the consistency requirements should be attenuated to eventual consistency so that the application can be designed in a non-blocking manner. One approach to solving problems of this scope is *optimistic replication*, as described by Shapiro et al. [101]. In optimistic replication, any client managing a replicated state is allowed to read or update the local replica at any time.

Such local updates are only tentative and their final outcome could change, as they may conflict with an update from another client. In optimistic replication, conflicts should not be visible to the user (since conflict resolution requiring human intervention would compromise consistency, which is undesirable in the intended use cases), so they must be resolved in the background, either by a decentralized protocol or a central server instance. The replicas are allowed to differ in the meantime, but they are expected to eventually become consistent. Figure 2.43 illustrates this behavior. Note that it only illustrates it for a single operation. In case of multiple operations, reordering the operations is also part of the conflict

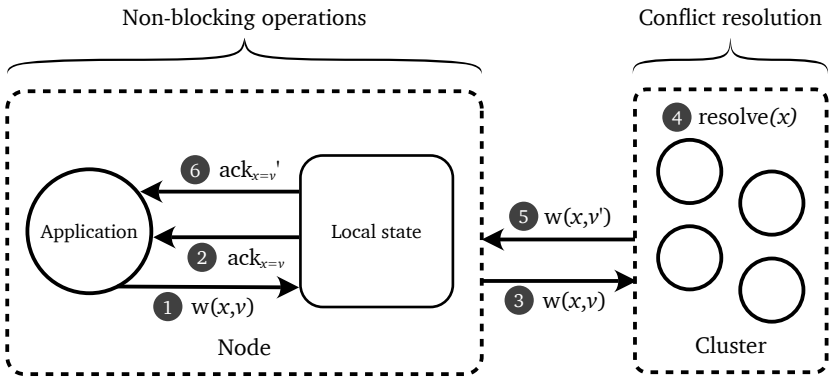


Figure 2.43: Schematic communication model between the nodes in optimistic replication for a single operation. A write is acknowledged immediately and non-blocking, without synchronization with other nodes. It is then eventually propagated to other nodes. In the event of a write conflict, the conflict is automatically resolved in the background using either a decentralized protocol or a centralized entity. Once the conflict is resolved, the actual agreed value is written back to the node's internal state and acknowledged.

resolution, depending on the protocol.

Optimistic replication is especially suitable for problems that meet the monotonicity requirements as described in the CALM theorem (cf. subsection 2.1.4.3), as this class of problems requires no conflict resolution at all.

Conflict-Free Replicated Data Types. An example of optimistic replication for strong eventual consistency are *conflict-free replicated data types* (CRDT). CRDTs were introduced by Shapiro et al. and describe data types with a data schema and operations on it that will be executed in replicated environments to ensure that objects always converge to a final state that is consistent across replicas [102]. CRDTs ensure this by requiring that all operations must be designed to be conflict-free, since CRDTs are intended for use in decentralized architectures where operation reordering is difficult to achieve. Therefore, any operation must be both *idempotent* (the same operation applied multiple times will result in the same outcome) and *commutative* (the order of operations does not influence the result of the chained operations), which also requires them to be free of side effects [22].

CRDTs take the consistency problem onto the level of data structures. They

[22]Recent research is trying to find such a replicated data type that does not require operations to be commutative. One approach describes *Strong Eventually Consistent Replicated Objects* (SECROs), aiming to build such a data type with the same dependability properties as CRDTs [103]. The authors achieve this by ensuring a total order of operations across all replicas, but still without synchronisation between the replicas: they show that it is possible to order the operations asynchronously. Another approach in

make use of optimistic replication to allow for such operations without the need of replicas to coordinate with each other (often allowing updates to be executed even offline) by relying on merge strategies to resolve consistency issues and conflicts, once the replicas synchronize again. They are designed in a way that the overall state resolves automatically, becoming eventually consistent. Due to the nature of the operations to be idempotent, CRDTs can only be used for a specific set of applications, such as distributed in-memory key-value stores like Redis, which is commonly used for caching [105]. Another common set of use cases are real-time collaborative systems, such as live document editing (relying on suitable data structures to hold the characters with good time complexity properties for both random inserts and reads, such as an *AVL tree* [106]) [107]. CRDTs are not suitable for append-only logs, streams or event stores, as designing an operation to append a record or event with idempotency and commutativeness is too expensive.

CRDTs are not the only way to achieve optimistic replication. Before CRDTs, *operational transformations* (OT) were the default approach for solving these kinds of problems, at least in academia. Due to their complexity and the difficulty to implement them and also to write formal proofs for different sets of operations needed in practical applications [108], they were often replaced by alternative approaches like CRDTs, so we will not discuss them in detail in this work.

In theory, CRDTs are designed for decentralized systems in which there is no central authority to decide the end state. In practice, there is oftentimes a central instance, such as in SaaS offerings on the web. Decentralized conflict resolution is no longer a requirement for those systems, so CRDTs could be too heavy-weight. Moving the conflict resolution to a central instance (which actually could be a cluster of nodes with stronger consistency guarentees) reduces complexity in the implementation of optimistic replication. This is actually the case in multiplayer video games—especially massive multiplayer online games (MMOGs). They introduce a class of problems that need techniques to synchronize at least a partial state between a lot of clients. Strong consistency would cause this games to experience bad performance drops, since a game's client wouldn't be able to continue until it's state is consistent across all subscribers to that state, so the way to go is eventual consistency. One can learn a lot from these approaches and adopt it to other real-time applications, such as Figma did for their collaborative design tool, comprehensively described in a blog post [109].

---

very recent literature that builds upon replicated coordination-free tree is available in a preprint by Nair et al. [104].

2.1.9.7 Coordination-Free Consistent Replication

Coordination-free consistent replication is similar to optimistic replication in terms of reducing the need for coordination between replicas, but it still provides strong consistency. We illustrate this approach by presenting a specific implementation called *Eris* that not only allows for coordination-free replication, but also coordination-free transactions [71]. The idea behind Eris and similar coordination-free protocols is to separate message ordering from reliable delivery in state machine replication. Eris was proposed and developed at Microsoft Research in 2017. What makes Eris so special and enables the elimination of coordination is a novel network infrastructure—which also means that this approach cannot be used in distributed systems deployed on general TCP-like networks.

Eris was designed to avoid the overhead of additional round-trips for atomic transactions in strong consistent systems. Traditional layered designs that use separate protocols for atomic transactions inter-shard and for replication of operations intra-shard introduce redundant coordination since the two layers are decoupled (see figure 2.29). Eris avoids this redundant coordination by unifying both replication and transaction coordination. Eris divides the responsibility for ordering and synchronisation between three parts: to ensure a consistent order of messages across shards (the basic requirement for strong consistency), it uses an *in-network concurrency control primitive* together with a *multi-sequenced groupcast*. To ensure atomicity and reliability of message delivery, the Eris protocol makes sure that transactions are processed either by all shards or none. As a result, it also avoids coordination for replication, not just for transactions: “it can process a large class of distributed transactions in a single round-trip from the client to the storage system without any explicit coordination between shards or replicas”.

Eris uses a quorum-based protocol to maintain safety. Similar to active replication (cf. 2.1.9.5), Eris clients send independent transactions directly to the replicas in the affected shards using multi-sequenced groupcast and wait for replies from a majority quorum from each shard.

Eris is based on *NOPaxos* [110]. NOPaxos extends Paxos by splitting sequencing (message ordering) and reliable message delivery. While the network layer is responsible for the first, the consensus protocol is responsible for the second[23]. In Eris, the sequencer in the network layer orders all messages and then sends the commits to every replica in the shards, and they directly acknowledge to the client.

[23] NOPaxos has been defined and validated in a TLA$^+$ specification: https://github.com/UWSysLab/NOPaxosTLA/blob/master/NOPaxos.tla

The *designated leaders* of each shard send the actual transaction results. Then, the designated leader is responsible for synchronous execution of its part of the transaction, while the followers log it and execute it later asynchronously. We illustrated the coordination sequence for the ideal case in figure . Compare it with figure 2.30, which demonstrates the coordination neccessary in traditional two layer models with 2PC and consensus protocols.

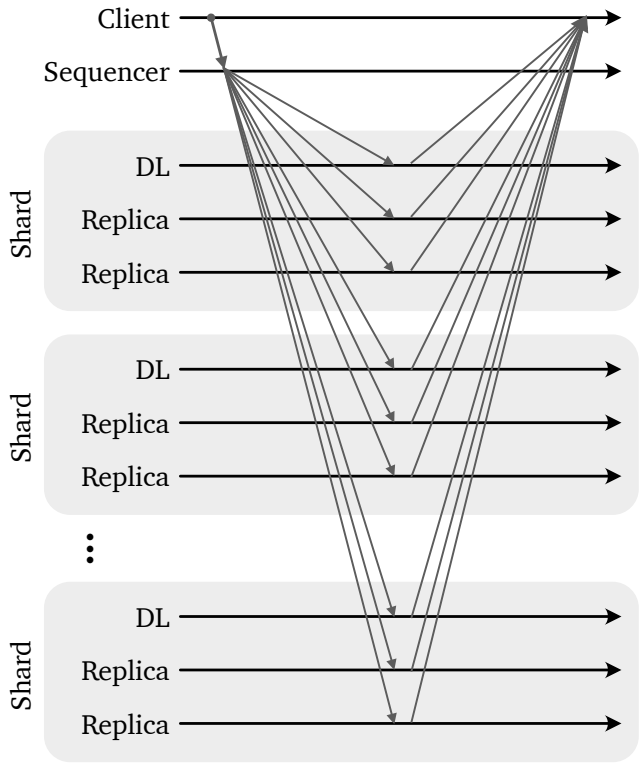


Figure 2.44: In the coordination-free transaction and replication protocol Eris, communication happens in a single round-trip in the normal case, while the transactions are consistently ordered by a sequencer on the network layer

Since one must be in charge of the underlying network infrastructure and protocol to run the sequencer, Eris cannot simply be used by any distributed system on generic TCP-like networks. Approaches such as Eris are generally found at large cloud providers such as AWS, Google, or Microsoft.

2.1.9.8 Chain Replication

Chain replication is a strongly consistent protocol for the fail-stop case. When it was introduced in 2014, it aimed to provide high throughput and availability without sacrificing strong consistency [111]. Chain replication allows to build a distributed storage system without an external cluster management process. The initial proposal for chain replication was targeting storage systems that sit between file systems and databases, such as key-value stores—and that is what chain replication is nowadays still used for. Chain replication is optimized for systems that offer

- A `store` operation to create and store objects,
- A `read` operation to retrieve a value from a single object,
- And an `update` operation to atomically change the state of a single object.

Chain replication extends the primary-copy approach, which itself can extend the state machine approach (cf. subsection 2.1.9.4). It is not using a quorum, but replicates objects to all nodes before an object is available for queries.

In chain replication, the nodes replicating a given object are ordered in a sequence to form a *chain*. The first nodes in the chain is called the *head*, and the last nodes is called the *tail*. The roles of these nodes are now the following (as illustrated in figure 2.45):

- The reply for every request is generated and sent by the tail.
- Each query request is served by the tail of the chain using the replica of the object stored at the tail.
- Each update request is directed to the head of the chain. The update command is executed atomically on the object replica at the head, then it forwards the resulting state changes to the next node of the chain. This process repeats until the command is executed at the tail.

This provides strong consistency since for a write to be acknowledged and to be served in a subsequent query, it must have passed every node in the chain first. At the same time, continuous writes are not blocked (thus, the system provides availability), since the head already committed the changes and is available to serve further writes.

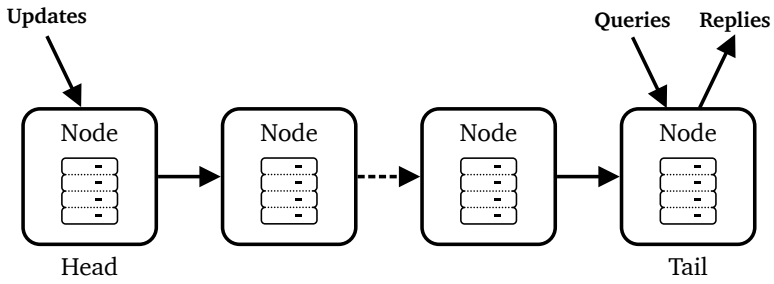


Figure 2.45: In chain replication, each write must pass every replica node before it is acknowledged and served in subsequent reads.

The interesting part here is the failure handling. A failing node renders the chain broken and must be mitigated accordingly. Once a node comes back, it must be inserted into the chain again, receiving all updates since its last failure. In chain replication, a master node is elected (in the initial proposal, it was elected using Paxos) to act as a failure detector. If the master detects a failed node (i.e., using heartbeats) it removes it from the chain and updates the chain, similar to a linked list.

Chain replication has some benefits, as well as some drawbacks. The head is the bottleneck of the chain for writes, since it provides write availability. Compared to quorum-based protocols, each node in the chain could act as the weakest link and slowing down serving of read requests. Hence, the failure detector must also detect slow nodes and handle them accordingly. Since only one node is allowed to answer read requests, chain replication can not scale with the number of clients and requests, and read latency does not decrease with the number of additional replicas in the chain, but actually increases.

Since the initial proposal of chain replication, deviations from the original strong consistency protocol appeared in the literature to mitigate the drawbacks. One of these proposals is *ChainReaction*, which provides causal+ consistency and geo-replication, and is able to leverage the presence of multiple replicas to distribute the load of read requests [112].

2.1.9.9 Summary

In this subsection, we have presented common theoretical replication protocols. We sum them up in table 2.9. In the next subsection, we take a look at practical variants of these protocols as they are implemented in real-life applications.

Table 2.9: Consistency levels of the presented replication protocols

Protocol	Consistency Level	Write Latency	Availability
Consensus	Strong	High	High
State Machine	Strong	High	High
ROWA	Strong	Highest	Lowest
Primary-Copy	Sequential-Strong	Low–High	Low–High
Active	Up to Sequential	Low–High	Low–Highest
Optimistic	Eventual	Low	Highest
Coordination-Free* *Only useable in rare cases	Strong	Medium	High
Chain	Strong	Low—High	High

2.1.10 Practical Replication Protocols

This subsection provides an overview of concrete, practical replication protocols to the reader. Some of them are used in production for years, while others are subject of academic studies only.

The following subchapter will then spend a closer look on Raft, a consensus-based state machine replication protocol. The author has chosen to use Raft for replication of the event store which is the subject of this thesis.

2.1.10.1 Paxos

One of the most popular distributed consensus algorithms is the Paxos consensus algoritm. It has been the de facto standard for distributed consensus in production systems for over two decades and is often used as a synonym for distributed consensus. Paxos is still widely deployed in production systems, such as several Google systems, including the Chubby Lock service [113] and the distributed NewSQL database system Spanner [84]. Most recent consensus protocols and most of the consensus literature are more or less based on Paxos or arose as a consequence of Paxos[24].

[24] One exception is the blockchain, which uses distributed, decentralized consensus protocols, see subsection 2.1.10.4.

History of Paxos. Paxos has been introduced by Leslie Lamport[25] in 1998 in the paper "The Part-Time Parliament" [115]. As allegorical as the title sounds, the paper also describes the Paxos algorithm in a picturesque approach. Just read the abstract to understand what we mean: "Recent archaeological discoveries on the island of Paxos reveal that the parliament functioned despite the peripatetic propensity of its part-time legislators. The legislators maintained consistent copies of the parliamentary record, despite their frequent forays from the chamber and the forgetfulness of their messengers. The Paxon parliament's protocol provides a new way of implementing the state machine approach to the design of distributed systems." The remainder of the paper is equally entertaining to read, but at the same time reveals a groundbreaking discovery in distributed consensus, but lacks enough detail to implement it. Due to the prevalence of Paxos, Lamport has attempted to make the algorithm more understandable and complete in his subsequent work [116], but it has been shown that people of all types (students as well as engineers) still have difficulty fully understanding it [117].

Further improvements and alterations on Paxos where made in the past, some by Lamport himself such as Fast Paxos [118], which introduces active replication characteristics (cf. subsection 2.1.9.5), and it has also been introduced for full transaction agreement in addition to a single value only [119].

Next to the Paxos consensus algorithm, there is the *Paxos algorithm*, which uses instances of the Paxos consensus algorithm in sequence to achieve fault-tolerant state machine replication. To reduce confusion, the first is referred to as *Basic Paxos*, while the latter is called *Multi-Paxos*.

Formal Verification. The Paxos consensus algorithm has been formally verified by Lamport using the TLA$^+$ formal specification language (see subsection 2.1.8 for reference)[118]; Multi-Paxos has also been formally verified by Chand et al. [120].

Consistency in Paxos. As for most consensus protocols, Paxos provides strong consistency through linearizability. Multi-Paxos uses the state machine replication approach, while the protocol does not put much emphasis on the state machine. Multi-Paxos executes the same set of commands in the same order, on multiple participants. Paxos fullfills the liveness and safety requirements of consensus protocols

[25] Leslie Lamport received the Turing Award in 2013 for all of his "fundamental contributions to the theory and practice of distributed and concurrent systems, notably the invention of concepts such as causality and logical clocks, safety and liveness, replicated state machines, and sequential consistency" [114]. Among other things, he authored, discovered and invented the LaTeX typesetting system, the TLA$^+$ formal specification language, Paxos, State Machine Replication and the Byzantine Agreement problem.

(see subsection 2.1.9.1).

Node Roles

In Paxos, nodes play different roles during the execution of the algorithm:

- **Proposers**: A node that tries to satisfy a client request by attempting to convince the acceptors to agree to its requests. In other protocols, they are often referred to as leaders or coordinators. Note that in Paxos, there can be multiple proposers. It is up to the implementation if there should be a single proposer only, or to have multiple (e.g., to support concurrency), which also introduces potentials for conflicts, which are handled by the Paxos algorithm.
- **Acceptors**: Nodes that listen and response to requests from proposers, able to form a quorum. Each message sent to an acceptor must be sent to and accepted by a quorum of acceptors.
- **Learners**: Additional nodes in the system which just "learn" the final values that are decided upon. They do not participate in a quorum of acceptors, but can still accept values that are replicated. Learners allows to reduce the read latency and availability of the system (i.e. in geo-replication), while they do not contribute to the safety, crash-fault tolerance and write availability of the system. Similar to a proposer once a learner receives a majority of accept messages then it knows that consensus has been reached on the value.

In practice, one node can have two or all these roles at the same time. From the point of view of the protocol, these roles are considered to be single, seperate virtual instances. For example, you could run a setup of 5 nodes, where 3 nodes act as acceptors (so a quorum of 2 can be formed), one of these nodes act as a proposer, and all the 5 nodes act as learners (therefore executing the actual operations after the agreement). The 2 nodes that do not act as acceptors do not contribute to the safety and fault-tolerance of the system, but this helps to improve the read latency of the system.

Protocol Phases

Due to the allegorical description of Paxos in the work of Lamport, there are multiple slightly different descriptions and implementations of the actual Paxos algorithm. Although they differ in detail, their outcome is the same. We present one of those.

Paxos works in two phases to make sure multiple nodes agree on the same value in spite of partial network or node failures. The phases are illustrated in figure

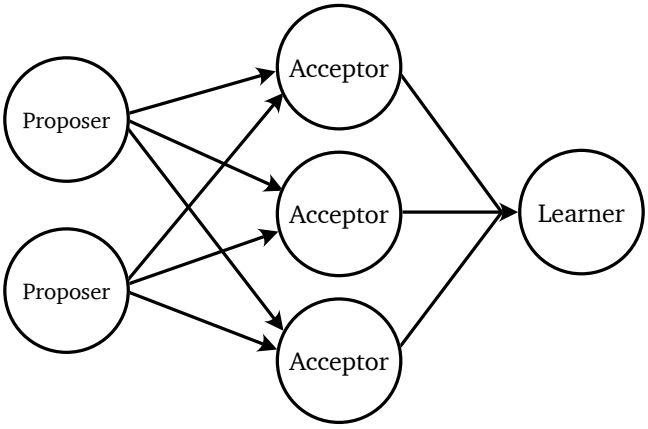


Figure 2.46: A common Paxos setup. Note that these are not neccessarily 6 different nodes, as one node can incorporate multiple roles at the same time.

2.47. These phases can go in multiple rounds per Paxos instance if some failure happen (such as network partitioning, node crashes, message timeouts or conflicting proposers, to mention a few). In Basic Paxos, a Paxos instance ends once the client request is satisfied.

Phase 1a: Prepare. Once a client requested a write, a proposer accepts the request. It then chooses a new proposal version number n (a generation clock, see subsection 2.1.9.2) and sends a `prepare`(n) request to all the acceptors.

Phase 1b: Promise. Acceptors receiving this request will compare that with the last proposal number it knows:

- If the received n is greater than any proposal number of a request they had already responded to, the acceptors responses with a promise. This promise comes with the guarantee that this acceptor will not accept any other proposals numbered less than n from now on. If a acceptor recently accepted a previous proposal during this paxos instance, it sends the previous proposal number n' and value v' of the latest accepted proposal in `promise`(n, n', v'), otherwhie it will just send `promise`(n).
- Else, the acceptor rejects the request, as it has already seen a higher proposal number, and sends a `nack` response to notify the proposer that it can stop the round.

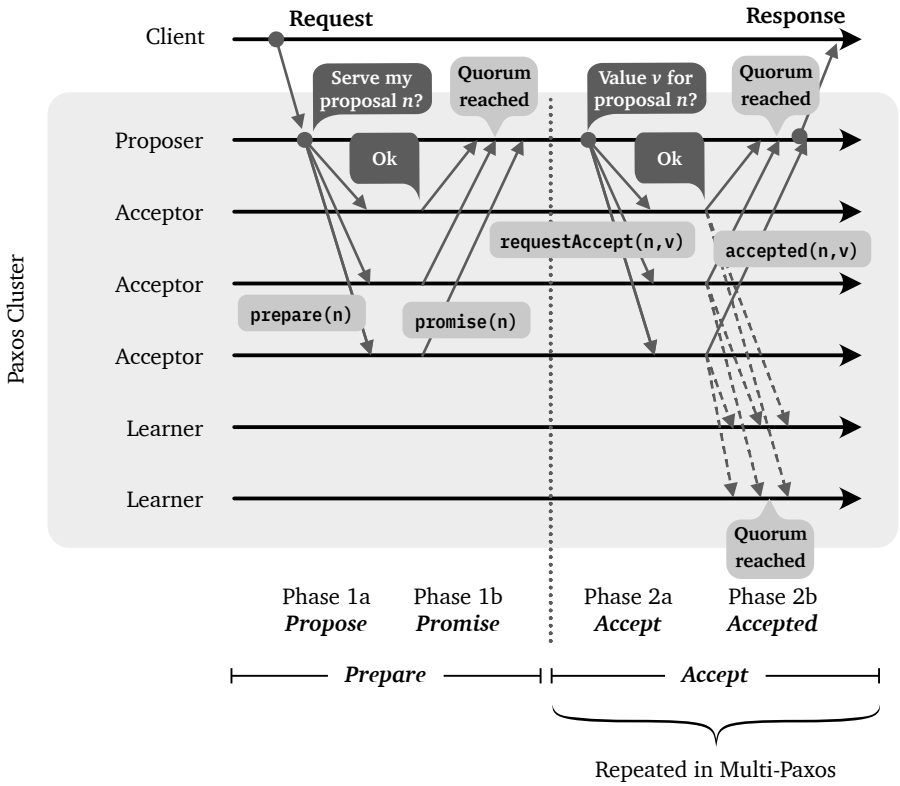


Figure 2.47: Idealistic message sequence in the Paxos consensus algorithm

Phase 2a: Accept. If the proposer receives promises from a quorum of the acceptors, then it will issue a request to accept a value `requestAccept`(n, v) with the proposal number n and the value to write v. The value to write is either - the value v' associated with the highest proposal number n' if any sent by the acceptors (to satisfy the consistency property that "at most one value can be learned" [118]), - else the original value it wanted to propose.

If the proposer does not get enough promises to reach a quorum in a certain timeframe, it restarts the whole process.

Phase 2b: Accepted. If an acceptor receives an accept request, it accepts the proposal unless it has already responded to a prepare request with a number greater than n. Whenever an acceptor accepts a proposal, it responds to the proposer and all the learners with `accepted`(n, v). Once the proposer receives `accepted`(n, v) from a quorum, it decides on the value v, writes it to its state and replies to the client.

Similarly, once a learner receives the command from a quorum, it also decides on this value.

There are also variants where there is an explicit third learning phase where not the acceptors talk to the learners, but the proposer sends a single `decide(`v`)` command to them after it decided on the value based on the quorum. This is illustrated in figure 2.48. Another variant requires the learners to reply to the client, not the proposer. By setting one node to be both a proposer and a learner, the same behavior as in the presented approach can be achievend. All thse variants have slightly different consequences, especially when it comes to network partitioning.

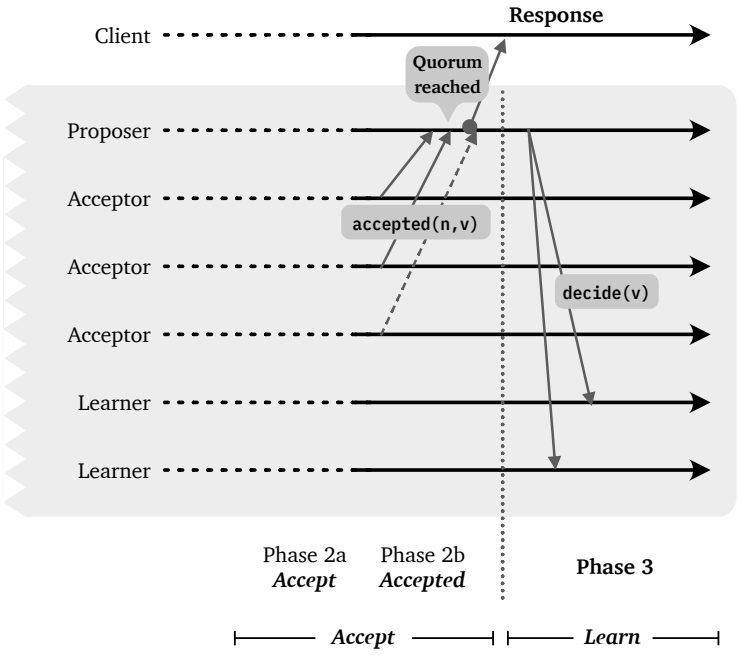


Figure 2.48: Alternative learn phase in the Paxos consensus algorithm

Multi-Paxos. Multi-Paxos[26] allows for log replication, which leads to state machine replication, by allowing to agree on a sequence of values rather than just a single value. It runs seperate instances of Basic Paxos for each entry in the log. Additional to the proposal number, it comes with a monotonically increasing index number

[26]Readers trying to find one proper specification of Multi-Paxos in acedemic literature will fail. It was not specified in detail in literature; the modern understanding of Multi-Paxos is derived from practical implementations in popular distributed systems (of vendors such as Google or Microsoft) and there are also differences in these implementations.

which is added to Prepare and Accept phase requests. Each single Paxos consensus instance can commit their values out-of-order, which makes the Multi-Paxos log working as follows: if a log is decided to a certain index, then all subsequently decisions will extend this log (i.e. the previously decided log up to the high-water mark is a common prefix). It is not guaranteed that the logs after the high-water mark are consistent in a quorum, though. They may have holes or even differing values at the same index, as long as they haven't been decided upon, as sketched in figure 2.49. This is an important characteristic: the state machine should only be served with the operations in the log up to the high-water mark, and client requests only replied to successfully once the log entry and all all previous log entries are decided. After the mark, the logs may only eventually converge. Not all acceptors logs need to know the whole log up to the high-water mark, as only a quorum is needed. The learners are responsible to provide the consistent log up to the high-water mark to the state machines.

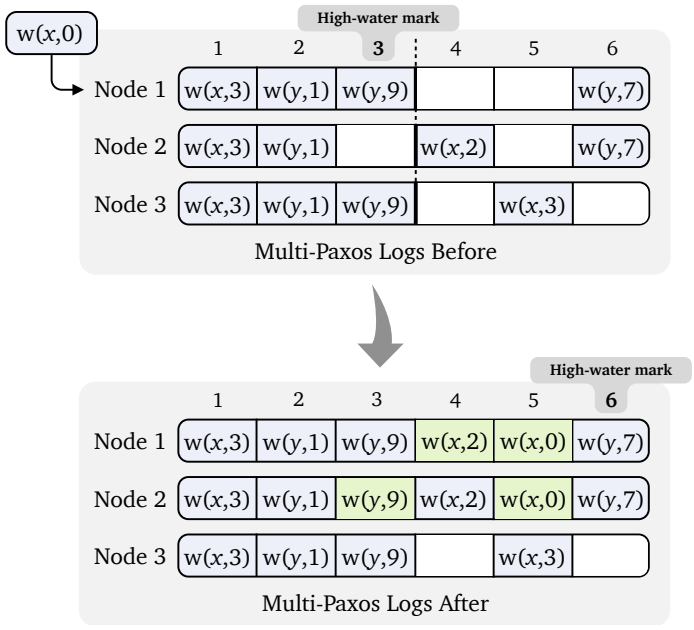


Figure 2.49: Logs in Multi-Paxos may have holes and differ after the high-water mark. Only a quorum of correct log entries is needed to achieve consistency. At each index, the logs are viewed separately.

In Multi-Paxos, phase 1 can skipped for subsequent requests as long as the cluster stays healthy, since the quorum already decided to accept values from one proposer.

This proposer can be decided to be a single leader, which reduces potential proposer conflicts. Phase 1 will be reintroduced once there is a conflict, which is detected in phase 2.

Why are there two phases at all? As long as the whole cluster stays healthy, the first phase could be omitted. But in case of a node crash, subsequent requests need to know what happened previously and if it was fully committed. Phase 1 is therefore needed to determine whether anything remains to be done. Phase 2 is then used to set the correct value to ensure consistency of all processes within the system. Due to the complexity of Paxos, this can be best understood by examples. We recommend that readers look for Paxos examples of failures to understand the algorithm and why it provides fault-tolerance.

2.1.10.2 ZooKeeper Atomic Broadcast (ZAB)

The *ZooKeeper atomic broadcast* (ZAB) protocol is the replication protocol that powers the ZooKeeper coordination service [121], which is one of the most popular open-source consensus systems. It was introduced by Yahoo! and later became an Apache project. It combines *atomic broadcast* and primary-copy (cf. subsection 2.1.9.4)with leader election. Atomic broadcast (also known as *total order broadcast*) is similar to consensus, but generalized for all types of message broadcasts in a distributed system. In fact, the consensus problem for crash-faults can be reduced to atomic broadcast and vice-versa [34]. Atomic broadcast puts the requirement on systems to "deliver the same messages in the same order".

ZooKeeper uses ZAB to propagate state changes throughout the ZooKeeper cluster [90]. It implements a primary-copy scheme in which a primary node serves and executes client requests and uses ZAB to propagate the corresponding state changes to follower nodes. ZAB guarantees total ordering (linearizability), i.e., when it delivers a change of state, all other changes on which it depends must be delivered first.

ZAB extends the atomic broadcast protocol by introducing a *primary order* property. It guarantees the correct order of state changes, even if different nodes act as the primary node over time, and allows multiple pending transactions. This allows ZooKeeper to buffer outstanding client requests and to send and execute prefixes of the buffer in batches.

Running ZooKeeper. ZooKeeper itself is an additional service that must be deployed as an external agent next to the actual system that should be coordinated, while ZooKeeper itself replicates its information with ZAB. This approach comes

with disadvantages: it adds additional maintenance efforts (a second service must be maintained, running on a cluster of seperate (virtual) nodes), reduces the overall availability since a system using ZooKeeper relies on it and becomes unavailable as soon as ZooKeeper is unavailable (see subsection **safety-reliability?** for reference), or during a network partitioning, and reduces the overall latency of the replicated system, as coordination with the ZooKeeper cluster adds an additional step. Thanks to understandable consensus protocols such as Raft (see subchapter 2.2), which also comes with a lot of open-source libraries ready to use, ZooKeeper lost a lot of its popularity over the last years, where it was replaced with embedded consensus solutions in major systems.

2.1.10.3 Viewstamped Replication

The *Viewstamped Replication* (VR) protocol is based on the primary-copy and state machine replication approach for fail-stop faults. It was initially developed by Oki and Liskov [122] in 1988 and later revisited by Liskov and Cowling in 2012 [123] for low-latency leader election and optimal strict serializability. While it was developed around the same time as Paxos, the auhors did not had any knowledge of Paxos. It differs from Paxos in that it is a replication protocol and not just a consensus protocol. It uses consensus to agree on a sequence of operations as part of a replicated state machine, with a consensus mechanism that is actually quite similar to Paxos. But in contrast to Paxos, VR does not require disk I/O when running an instance of the consensus protocol.

Viewstamped Replication is divided into four sub-protocols that describe

- How client requests are handled,
- How the group reorganizes when a primary node fails,
- How a failed replica is able to rejoin the group,
- And how to reconfigure the network in case of a new node joining or a node leaving the group.

We outline these protocols briefly in the next paragraphs, but we won't go into the details.

Request Handling. Client nodes run their requests through a client library (the VR proxy). The proxy then communicates with the replicas to run the operation on a quorum and returns the result to the client. VR uses a primary replica to order client requests; the other replicas act as copies that simply accept the order selected by the primary. A request is served once it is acknowledged by a quorum of the copies.

Primary Failure. In case of a failure of the primary node, a new node is selected to be the next primary. This starts a new *view*. A view is a generation clock that denotes the phase of a particular node configuration. The system moves through a sequence of views over time. The primary is continously monitored by the copy nodes through heartbeat detection, and if it appears to be faulty, they instantiate a view change protocol to select a new primary.

Node Recovery. When a replica recovers after a crash it cannot participate in request processing and view changes until it has a state at least as recent as when it failed. To restore the state of a crashed node, a recovery protocol is instantiated. After the recovery protocol, the recovering node is able to participate in the other protocols again. During the recovery protocol, the recovering node requests a recovery response from a quorum. The nodes respond with the current number of the view that they know and in addition, the primary answers with its full log. Only if a quorum of nodes replied with the same view number, the recovering node applies the log it received from the primary.

Network Reconfiguration. To support network reconfiguration, next to the view number, an incrementing epoch number is introduced. The epoch number denotes the epoch of a network configuration. Nodes only process messages that match their known epoch number. If they receive a message including a higher epoch number, they move to this new epoch. When the reconfiguration protocol is instantiated by a client, a new epoch number is filed and the nodes set their status to *transitioning*. Nodes switch their status to *normal* once they have received the complete log up to the start of the new epoch. Nodes that are being replaced only shut down once they ensured that their log has been transferred successfully.

2.1.10.4 Blockchain Consensus Protocols

In recent years, the problem of byzantine fault tolerant consensus has raised significantly more attention due to the widespread success of blockchains and blockchain-based applications such as cryptocurrencies. Blockchains are distributed systems par excellence. In this section, we will cover briefly the realm of blockchains and how replication and consistency in blockchains work. We do this so the reader can better understand the requirements on modern-day replication protocols, but we won't go in-depth, since this would exceed the scope of this work. To discuss replication in blockchains, we must differentiate between permissioned and permission-less blockchains first [124].

Permissioned vs. Permission-Less Blockchains. In blockchains, strong consistency describes that all nodes have the same *ledger* at the same time. During updates of the distributed ledger, any subsequent requests will have to wait until this update is committed. This is handled differently in permissioned and permission-less blockchains.

While permissioned blockchains are generally operated by and inside closed organizations (or across multiple authorized organisation) on a manageable number of nodes, permission-less, public blockchains run on a much bigger scale, such as those for cryptocurrencies. Due to their permission-less property, everyone can participate in such a network without prior authorization, which makes byzantine-fault tolerance an important property in such blockchains.

Hyperledger Fabric. One example for permissioned blockchain systems is *Hyperledger Fabric*. We show this here to demonstrate the different requirements and usage of replication protocols between permissioned and permission-less chains. Hyperledger Fabric is a modular and extensible open-source system for implementing and operating permissioned blockchain implementations [125]. It relies on an ordering service for strongly consistent ordering of blocks. Previously, it used Apache Kafka (cf. subsection 3.2.2.6) as this ordering service, since it did not implement this service itself. As of version 1.4.1, it implements the ordering service itself using the implementation the Raft consensus protocol (cf. subchapter 2.2) from the etcd project [126]. The use of Kafka has been deprecated since version 2. The problem scope of permissioned blockchains is similar to that of other distributed data storage systems that manage immutable, append-only data, such as event stores (see subchapter 2.3).

Permission-Less Blockchains. When asking someone about blockchains, most people will think about cryptocurrencies and other applications that run on permission-less blockchains. So far, we only covered practical protocols for crash-fault tolerance in distributed, but not decentralized networks. These protocols are not applicable in decentralized networks with a huge amount of nodes, such as in *permission-less blockchains*. As blockchains typically have hundreds to hundred of thousands of replicas and need to provide high levels of consistency, classic single-leader state machine protocols are not applicable here, as they will dramatically slow down the whole chain due to their cost of replication. In addition, crash-fault tolerance is a small problem in permission-less blockchains, compared to byzantine faults. In decentralized networks, nodes can join and leave the network at any time, and this

should happen without reducing the availability of the network. Byzantine-faults are a major threat to such a network since they are not permissioned, which means that any node can participate in the consensus process, able to expose any arbitrary or faulty data in the consensus process. Since permission-less blockchains try to find consensus in critical applications such as cryptocurrencies without a central authority, they must be tamper-resistant. As we have shown in subsection, consensus in the byzantine case requires huge amounts of coordination between nodes, which is not feasible in networks with so many nodes.

With the advent of such blockchains, the amount of literature on byzantine-fault tolerant consensus protocols exploded, and is still a topic undergoing very frequent changes and innovations. This has led to an overload of the term *consensus protocols*, since some literature will only return blockchain protocols for this term. To mention a few of the most popular, there are protocols like *practical byzantine-fault tolerance* (pBFT), *Proof of Work* (PoW, with the first one the *Nakamoto consensus protocol* [127]), and *Proof of Stake* (such as in Ethereum 2.0). We refer to [93] and [128] for an overview of more such protocols.

To reduce the amount of coordination needed between nodes—which will otherwhise grow exponentially—many of these protocols rely on *probabilistic consensus algorithms* which only guarantee eventual consistency, but to a high degree of probability [129]. In most cases, these algorithms provide practically strong consistency, but they are still prone to fatal faults, such as network partitioning (also known as a *ledger fork*), where different nodes in the network have a different view of the accepted order of blocks. Some of these non-agreed forks in the past led to the creation of a new cryptocurrency, such as in the example of *Bitcoin Cash*. Furthermore, the consistency discussion has sparked controversy. For instance, some argue that the Nakamoto consensus protocol of the Bitcoin network provides eventual consistency only, while others claim that Bitcoin guarantees strong consistency [130].

2.1.11 Summary

In this chapter, we have shown that replication always involves tradeoffs. There are several theorems that state that you cannot have consistency at the same time with full availability and low latency. Several important consistency models have been discussed, including their tradeoffs and limitations. It was shown that the limitations strongly depend on the use cases of the replicated data store. Finally, relevant replication protocols were discussed with respect to these consistency models, both from a theoretical and practical point of view. We presented protocols that are

heavily used in production systems such as Paxos, CRDTs, blockchain consensus protocols or Viewstamped Replication, as well as historic protocols such as ROWA and primary-copy.

In the next chapter, we will take a closer look at one specific protocol for replicating state machines with strong consistency that yields interesting properties for our work: the Raft consensus protocol.

2.2 Raft, an Understandable Consensus Algorithm

> There is a race between the increasing complexity of the systems we build and our ability to develop intellectual tools for understanding their complexity. If the race is won by our tools, then systems will eventually become easier to use and more reliable. If not, they will continue to become harder to use and less reliable [...].
>
> — Leslie Lamport

In this subchapter, we present the Raft consensus algorithm[27]. First, we explain the motivation behind Raft. Second, we provide a general overview of the components and behavior of a Raft network as described by Ongaro and Ousterhout in the paper [117] and the dissertation [88]. The protocol is then explained in detail. Since Raft belongs to the family of state machine replication, we refer to section 2.1.9.2 for the basics of the protocol. Finally, we discuss some recent extensions of the Raft protocol.

Nowadays, Raft is used in many production systems, such as

- Permissioned blockchains (e.g., Hyperledger Fabric, see subsection 2.1.10.4),
- Event and message brokers (e.g., Kafka and RabbitMQ, see subsection 3.2.2.6),
- NoSQL object stores (e.g., MongoDB [131]),
- Distributed relational databases (e.g., CockroachDB [132]),
- Process orchestration systems (e.g., Camunda Zeebe [16]),
- Key-value stores (e.g., etcd [19]),
- Container orchestration (e.g., Kubernetes [14]),
- smart grids [133] and many more.

[27] The name of this protocol is a metaphor and is derived from the fact that it is built out of replicated logs, much like a real raft is built from multiple wooden logs.

2.2.1 Understandability

> There is a race between the increasing complexity of the systems we build and our ability to develop intellectual tools for understanding their complexity. If the race is won by our tools, then systems will eventually become easier to use and more reliable. If not, they will continue to become harder to use and less reliable for all but a relatively small set of common tasks. Given how hard thinking is, if those intellectual tools are to succeed, they will have to substitute calculation for thought.
>
> — Leslie Lamport

Why is there a need for another consensus protocol which offers similar fault-tolerance and performance characteristics as Paxos (see subsection 2.1.10.1)? The Raft consensus protocol was presented in 2013 by Diego Ongaro in reaction to the complexity of Paxos. Paxos was (and still often is) the default protocol taught in lectures at universities when it comes to consensus protocols. The authors of Raft observed that despite its popularity, Paxos is an algorithm that is difficult to understand and implement, especially since all official publications about Paxos lack a complete description of the protocol. We also discovered and described the latter in subsection 2.1.10.1. This inhibits progress in these areas and hampers discourse. Raft is a consensus protocol that was explicitly designed as an alternative to the Paxos family of algorithms to solve this understandability issues of how consensus can be achieved.

There are some other main differences to Paxos and Viewstamped Replication in the protocol, which are handled in sections 2.2.2 and 2.2.3.

In essential, Raft belongs to the same class of protocols as Paxos, as well as Viewstamped Replication (see subsection 2.1.10.3): state machine replication. But Raft adds understandability to make consensus available to a broader audience. Since complexity is often a reason why research or development projects are slowed down or even canceled, Ongora picks up the term *complexity budget*: “Every system has a *complexity budget*: the system offers some benefits for its users, but if its complexity outweighs these benefits, then the system is no longer worthwhile” [88]. This saying, the complexity of an algorithm or protocol limits how much of it can fit into the heads of academic researchers, distributed systems designers, and engineers. In his dissertation, Ongora also expresses criticism of the academic community to “not consider[...] understandability per se to be an important contribution”. To support understandability and to teach the protocol, the Raft authors even implemented an

interactive visualization of the Raft consensus protocol, called *The Secret Lives of Data* [134].

The understandability was evaluated in a user study and the results of this study confirmed that Raft is more understandable as Paxos. The study compared the answers of students to quiz questions about Raft and Paxos after they learned each algorithm. We refer to the dissertation for the results [88]. As another result to the user study, it is also easier to verify the correctness of the Raft algorithm than that of Multi-Paxos. Attention—This is the opinion of the author of this work: The Raft TLA$^+$ specification[28] looks more understandable than the Paxos[29] and Multi-Paxos[30] specifications.

2.2.2 Main Differences to Paxos

The main differences between Raft and Paxos are listed in this section.

Complexity. We have already laid out the differences in complexity between the two protocols.

The Problem Perspective. While Paxos originated from the distributed consensus problem, Raft originates directly from the perspective of a replicated state machine. The distributed consensus problem focuses on agreeing to a single value. Multi-Paxos is still derived from this perspective, making it difficult to understand and argue about the ordering guarantees of Paxos for multiple operations. In contrast, strongly consistent ordering sits in the core of the Raft consensus protocol, making it easy to understand and reason about.

A Clear Definition of the Algorithm. While the multitude of different variants of the Paxos protocol family lack a single, clear definition of what to implement—especially for Multi-Paxos which is mostly derived from practical implementation in the software industry—Raft provides a complete and concise specification of every aspect of the protocol that can be easily followed to implement Raft in practice.

Consistent Logs. While logs in Paxos can have holes, Raft logs can't. Raft logs grow monotonically and append-only; their high-water mark shows to the latest

[28] https://github.com/ongardie/raft.tla/blob/master/raft.tla
[29] https://github.com/tlaplus/tlapm/blob/main/examples/paxos/Paxos.tla
[30] https://github.com/DistAlgo/proofs/blob/master/multi-paxos/MultiPaxos.tla

committed log entry, and after that, they can differ in length, but always have a common prefix. In contrast, the Multi-Paxos log comes with nondeterminism; each log slot is decided independently which leads to holes in the logs in different places, which can block applications which require complete, consistent logs; the high-water mark in Paxos may move slower than the one in Raft.

Less Complex Roles. In Raft, there is no seperate learner role. Each node takes only one role at a time, while in Paxos, a node can have all possible combination of the three roles. This reduces complexity of every system running Raft, since there are no varying network topologies.

This comes with the sacrifice of having seperate nodes acting solely as learners to help distributing the replicated data, e.g. to increase data-locality for geo-replication, and to improve the read latency, without sacrificing write latency.

Strong Single Leader. The concept of multiple proposers in Paxos introduces complexity and confusion, making the algorithm difficult to understand and implement. For many practical use cases, a single leader is often sufficient, so Raft is more suitable in these cases. Raft elects one node to be the leader. Subsequently, all requests go through the leader. This simplifies the management of the replicated log. These two basic protocol steps are separate, whereas in Paxos these complementary protocol steps are intertwined.

But, there are also extensions to Multi-Paxos, describing a *distinguished leader*, which acts at the same time as a single distinguished proposer and learner, responsible for serving all client reads and writes, which is similar to Raft.

Table 2.10: Differences between Raft and Multi-Paxos

	Raft	**Multi-Paxos**
Leader	Strong	Weak; many proposers allowed
Log replication	Monotonicity guaranteed	Holes allowed
Log submission	Linearized commits	Asynchronous commits
Phases	Leader election & Log replication	Prepare phase & Accept phase

2.2.3 Main Differences to Viewstamped Replication

Similar to the previous section, this section shows the main differences between Raft and Viewstamped Replication (VR, see subsection 2.1.10.3).

Complexity. Viewstamped Replication is probably the protocol with a specification closest to that of Raft. But still, it is more complex than Raft. Especially since Raft provides strong leader properties and minimizes the functionality in non-leaders, its protocol is more compact and less complex than that of VR, and also offers a more complete specification: the Raft protocol only uses 4 different message types, whereas VR needs 10 to perform the same tasks.

Leader Election. Raft guarantees that if a leader is elected, its log contains all of the committed entries, making it easier to reason about the Raft logs in the cluster. In Raft, log entries only flow in one direction: from leaders to followers. Additionally, leaders will never overwrite existing entries in their own logs. Compared to that, VR maintains additional protocol steps to identify missing log entries in primary nodes and to transmit them to the new primary, which adds complexity both in the protocol and in intermediate cluster states. There is also a difference in the way a leader is voted for: in VR, a node accepts a voting request using a deterministic function of the term number, while in Raft, a node accepts a voting request in a non-deterministic way by accepting the vote of the first node who asks.

Transferring Logs. Raft uses term-index pairs to denote the current state of the replicated logs and to test for consistency, whereas VR transfers entire logs, such as in leader election.

Primary-Copy. Viewstamped Replication originates from the Primary-Copy approach, while Raft puts the log first. This means that in Viewstamped Replication, state changes are replicated rather than the original commands.

For a more detailed explanation of the differences, we refer to the dissertation [88] and this discussion with Ongaro in the Raft-Dev mailing group [135].

2.2.4 Protocol Overview

The Raft protocol implements a replicated state machine by managing a replicated log. As a result, Raft provides strong consistency. Raft is a protocol that is as concise as possible without lacking correctness and including all properties that

are important in state machine replication. It divides the problem of multi-value consensus into three relatively distinct subproblems:

- Leader election,
- Log replication,
- And safety (cf. section 2.1.2 for a definition of safety).

We will outline these three subproblems throughout the next sections. For a full overview of the protocol, we refer to the the original paper [117] and the dissertation [88]. We also extracted the summary page of the protocol from the original paper into the appendix of this work.

Consistency. Raft is linearizable, thus providing strong consistency, since it handles all reads and writes by a single, strong leader. From a trade-off perspective, Raft is a CP class (in CAP) respectively PC/EC class protocol (in PACELC).

Replicated State Machine. We presented the state machine replication approach in subsection 2.1.9.2. Raft implements all important properties of state machine replication:

- Leader election,
- A generation clock (called a *term*),
- A high-water mark for the replicated log (called the `lastApplied` term-index),
- Log truncation,
- Snapshotting and state transfer,
- And network reconfigurations (cluster membership changes).

The following subsections describe what is specific to Raft only, and for everything else we refer to the descriptions in subsection 2.1.9.2.

Messaging. The basic Raft protocol gets by with just two messages: `appendEntries` and `requestVote`. Raft also supports log compaction and cluster membership changes with two additional messages: `installSnapshot` and `add/removeServer`.

2.2.5 Node Roles

In Raft, a node has one of three roles, and only exactly one role at the same time. Figure 2.50 shows these roles and how a node transitions between them.

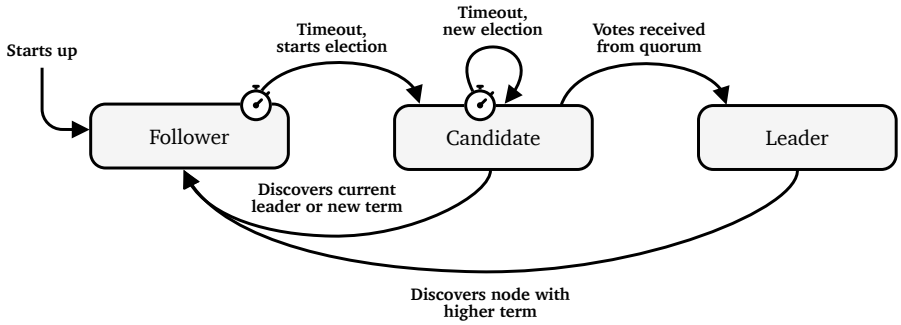


Figure 2.50: Transitions between roles of a cluster node in Raft. When a node joins the cluster, it starts as a follower. After a random timeout, it starts an election by voting for itself, turning into a candidate. To add pseudo-synchronous behavior to mitigate the FLP impossibility problem, it starts a new election after the timeout. Every new election introduces a new term, closing the previous one.

Follower. All nodes begin in the follower role on Raft cluster start up or if a new node joins the cluster. A follower listens to `appendEntries` requests from the leader. If such a request contains log entries, it applies them to its log. It also commits all entries in its log that are older then the `lastApplied` term-index pair in this message. It also interprets `appendEntries` requests as heartbeats from the leader. Therefore, it can also receive `appendEntries` requests without log entries in the payload. Once it does not receive a heartbeat from the leader after a certain *election timeout* period, it assumes the leader node crashed (failure detection), transitions into a candidate, and starts a leader election.

Candidate. As a candidate, a node votes for itself and sends `requestVote` messages to all other nodes in the cluster. If it receives granted votes from a majority of votes of the total cluster (including itself), it transitions into the next leader.

Leader. For a Raft cluster, only one single leader is allowed at a time. The leader is the only node that is allowed to respond to client requests, whether they are read or write requests. The leader sends `appendEntries` requests to the followers to replicate the write requests they receive from clients. When on idle, the leader continues to send `appendEntries` with an empty payload which acts as a heartbeat.

When the network is partitioned, a split-brain situation can emerge. Only in this situation, there can be more than one leader, since both partitions won't know if the other is still alive and assumes the nodes of it do be crashed. The Raft protocol ensures that after a cluster recovers from partitioning, a single leader is elected and

strong consistency is restored again across all nodes, which may result in loss of data at one of the partitions.

2.2.6 Guarantees

Raft comes with several guarantees, that in turn guarantee liveness, safety and strong consistency of the protocol. These guarantees are given by the definition of the log replication and leader election steps, which we will present in the subsequent sections. The guarantees have been proven in the Raft dissertation [88], and can be verified using the Raft TLA$^+$ specification. We reproduce these guarantees in their original form here.

Table 2.11: Raft guarantees, as described by Ongaro and Ousterhout

Election Safety	At most one leader can be elected in a given term.
Leader Append-Only	A leader never overwrites or deletes entries in its log; it only appends new entries.
Log Matching	If two logs contain an entry with the same index and term, then the logs are identical in all entries up through the given index.
Leader Completeness	If a log entry is committed in a given term, then that entry will be present in the logs of the leaders for all higher-numbered terms.
State Machine Safety	If a server has applied a log entry at a given index to its state machine, no other server will ever apply a different log entry for the same index.

2.2.7 Log Replication

In Raft, a log entry contains the write command that is to be replicated and applied to the state machine, the current term and the index of this log entry. This is illustrated in figure 2.51. The term is a monotonically growing generation clock (cf. subsection 2.1.9.2), representing the current phase of an incumbent leader. With every leader election, a new term is started, increasing the term number. The term number is crucial in providing safety during log replication and leader election in the face of failures. If a follower node that is in a higher term receives messages from a lower term, it rejects these messages and requests a new voting.

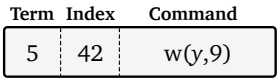


Figure 2.51: Anatomy of a Raft log entry. A Raft log entry contains the term and the index of its creation, as well as the command payload.

Every node maintains a `lastApplied` and `commitIndex` term-index pair. The `lastApplied` term-index pair denotes the last log entry that was applied to the state machine, while the `commitIndex` denotes the last log entry that is known to be *committed*. An entry is considered committed once it can be safely applied to state machines (i.e., it is known to a node quorum). Hence, the `commitIndex` implements the high-water mark of state machine replication.

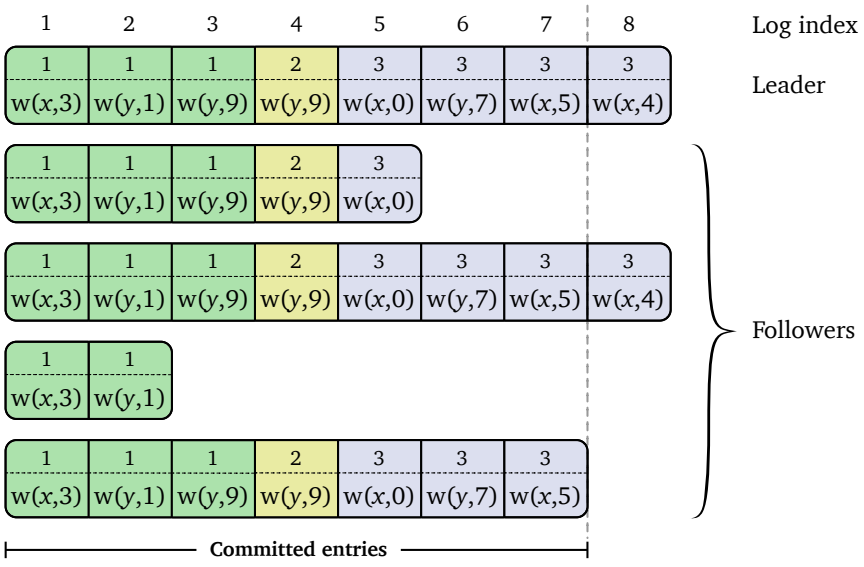


Figure 2.52: Raft logs of different nodes of a cluster. Every log entry contains the current term and index as well as its command. Entries are committed if they have been acknowledged by a quorum, thus safe to be applied to the state.

In figure 2.53, the replication of a log entry is illustrated[31] as it happens in the case of normal operation.

[31] The illustration is based on a blog post by Martin Fowler: https://martinfowler.com/articles/patterns-of-distributed-systems/replicated-log.html

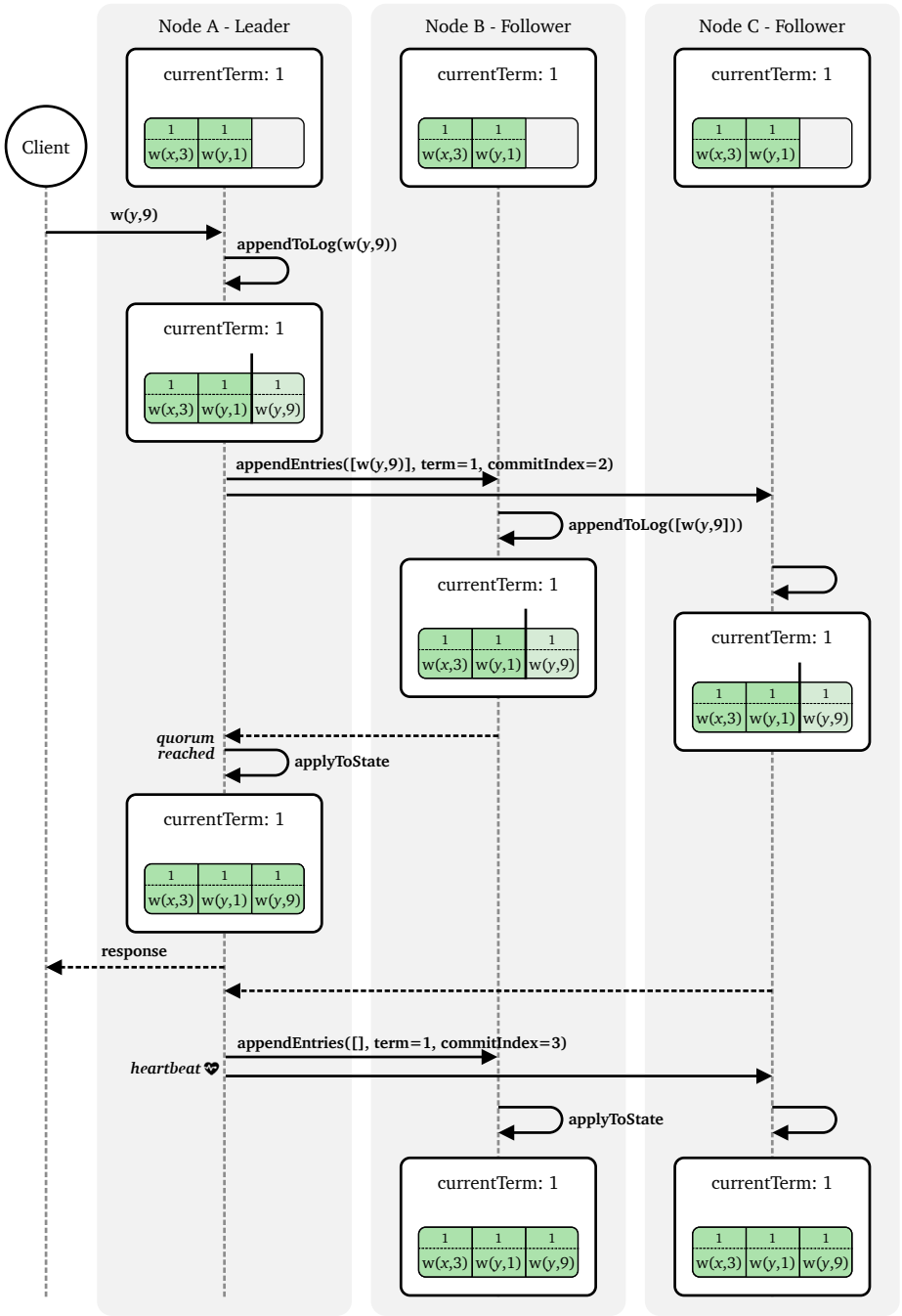


Figure 2.53: Simplified illustration of the log replication in Raft, based on Martin Fowler

A client sends a write request to a Raft cluster of three nodes. All nodes are already in a consistent state, containing the same entries in their logs. Once the write request arrives at the cluster leader (Node A), it appends it to its own log. Subsequently, it requests the other nodes to append this entry by sending a `appendEntries` message. This message contains the entries to append (here, a single entry), as well as the current term of the leader and the last `commitIndex`.

The followers receiving this message compares the term and `commitIndex` with their own logs. In case one follower contains more log entries from the same term, it rejects the request and starts a new vote. If it contains uncommitted entries from an older term—which could have happened due to network partitioning—it removes them from its log and continues accepting the append request. If the term in the message is lower than the one of the follower, the follower rejects and notifies the leader, which steps back into the follower role. In the case of the illustration, everything is ok, so the followers accept the request and append the entry to their logs.

The followers acknowledge the append. Once the leader receives acknowledgements from a quorum (including itself), it increases its `commitIndex` to the highest term-index pair of the acknowledged entries. It is now safe to append the entries to the state machine. In the next `appendEntries` request—in this case a heartbeat—it sends the updated `commitIndex`. The followers then update their own `commitIndex`, which also allows them to append the previously replicated log entries.

This process ensures a global execution ordering of commands to the replicated state machines.

2.2.8 Leader Election

We described the node roles in section 2.2.5. When a node joins the cluster, it joins as a follower. After a random election timeout, it starts an election by voting for itself, turning into a candidate. To add pseudo-synchronous behavior (to mitigate the FLP impossibility problem), it will restart the election again if no leader was decided after another random timeout. The randomness is important here to avoid deadlocks and to introduce nondeterminism, which is also required due to the FLP problem (cf. paragraph 2.1.9.1). Leader election is the part of the Raft protocol where timing is most critical. The leader election timeout is to be chosen in a way that in general, after a successful vote, heartbeats are sent before other followers time out and start a vote. This is reflect in the following timing requirement:

$$\texttt{broadcastTime} \ll \texttt{electionTimeout} \ll \text{MTBF}$$

Here, the `broadcastTime` is the average time it takes to send messages to all nodes in the cluster and receive their responses, and MTBF is the Mean Time Between Failures (cf. section 2.1.1).

Every new election introduces a new term, closing the previous one. This increases the term number. If a candidate node receives votes from a quorum (an absolute majority of the nodes), including its own vote, it immediately turns into a leader, sending out heartbeats to notify all other nodes of the successful vote. If due to network latency, partitioning or other problems a new leader node discovers any node with a higher term then the own one, it turns back into a follower to potentially start a new vote for a new term, to restore consistency. The same applies to candidates. If any candidate receives a heartbeat from a newly assigned leader, it turns back into a follower. This ensures that there is only one leader for any given term—at least if there is no network partition[32]. The way Raft is designed ensures that a quorum always knows the current committed log entries. This results in a guarantee that during voting, a candidate will not turn into a leader unless its log contains all committed entries. We illustrate the voting phase by an example in figures 2.54—2.56.

In figure 2.54, nodes D and E crashed. Node E was the leader in term 1. Since there is no leader, nodes A—C do not receive a `appendEntries` message anymore. Node B times out, turns into a candidate and starts a leader election for a new term, voting for itself and sending `requestVote` messages to the other nodes. The `requestVote` message contains the current term and the last known term-index of node B. Since it contains less entries than Node A, its vote is rejected by A. Node B does not receive a grant from a quorum (at least 3 nodes, since the original cluster contained 5 nodes) and hence loses this vote, transitioning back into a follower.

[32]This leader election protocol is similar to the view change protocol of Viewstamped Peplication, where a term is called a view.

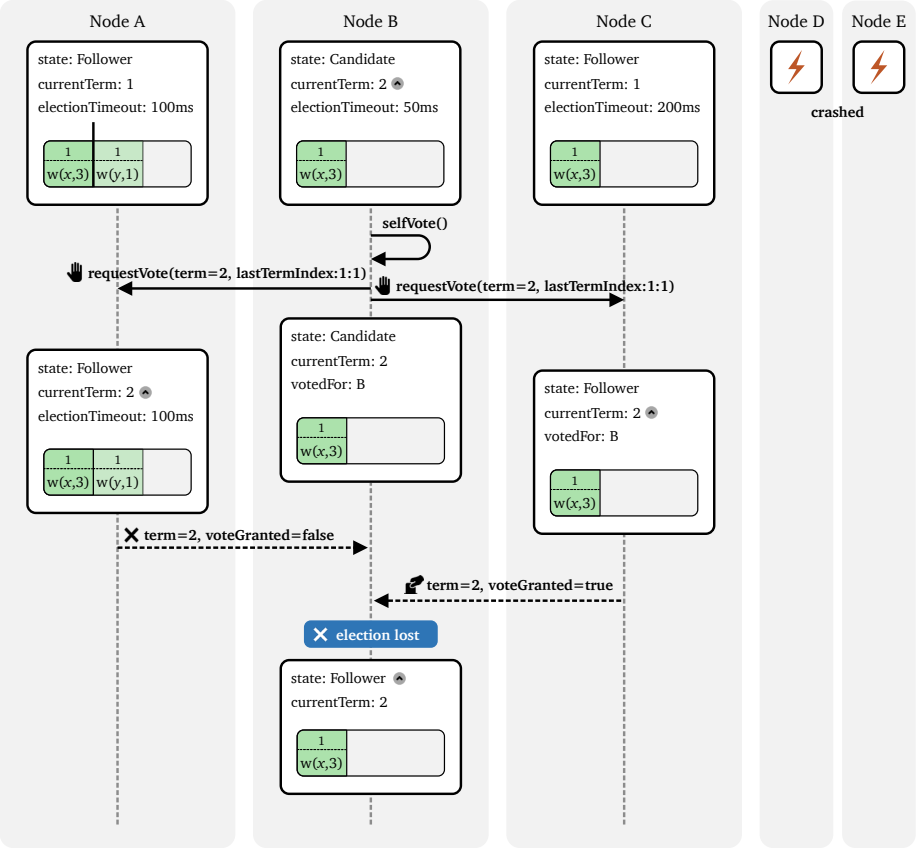


Figure 2.54: Illustration of the leader election in Raft after a node failure. A node requests a vote, but fails the vote since it does not receive a grant from a majority of cluster nodes (including itself).

After the failed election, node A times out and starts a new one. It increases the term again, now to 3, votes for itself and requests votes. Since it contains all known log entries and there are no other issues, it wins the election, turning into the new leader for term 3. It then announces its new leadership by sending `requestVote` messages to the other followers, preventing them from starting a new vote, and also sending the remaining log entry in the payload of the message. This entry is then appended to the logs of the followers.

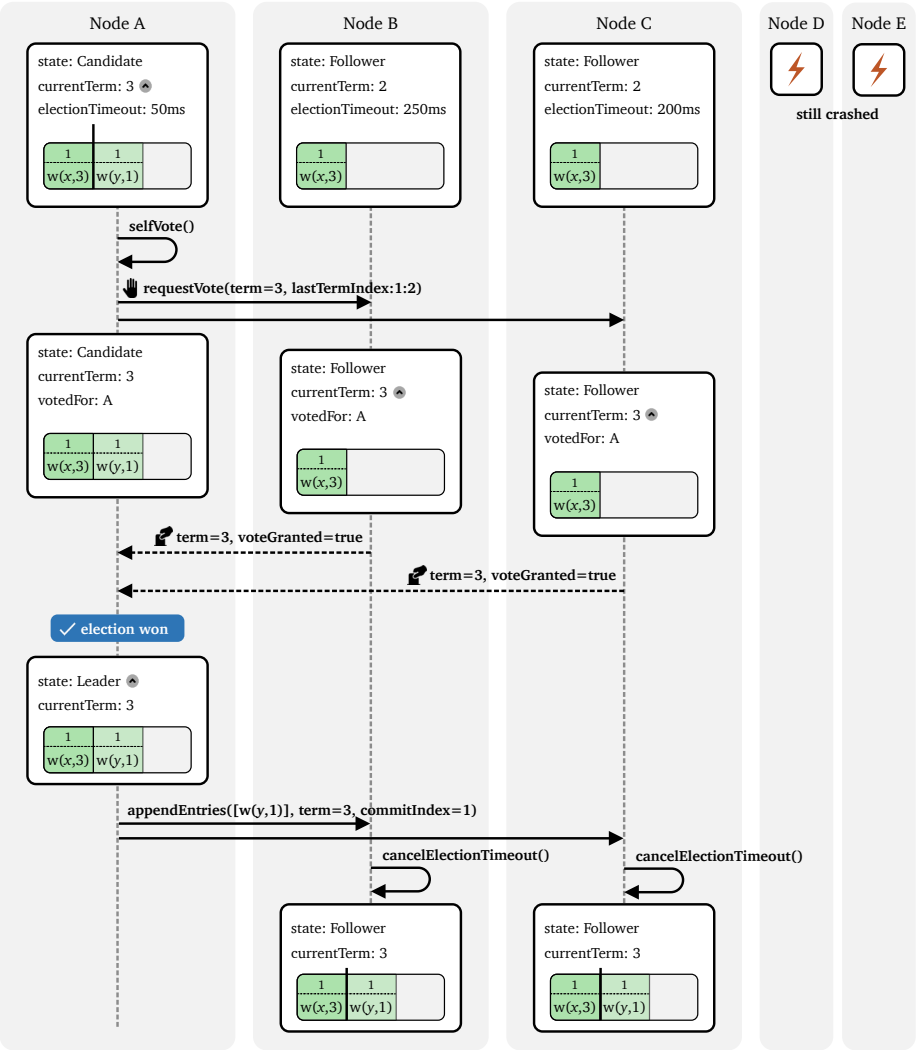


Figure 2.55: A successful leader election in Raft. A node requests a vote and receives a grant from a majority of cluster nodes.

Node A replicated the second entry from term 1 to nodes B and C. Since this entry is from a older term, the entry is not committed after this replication. To commit this entry and apply it to the state machine, a no-operation is appended, replicated and committed in the next heartbeat to update all Raft logs to the newly started term.

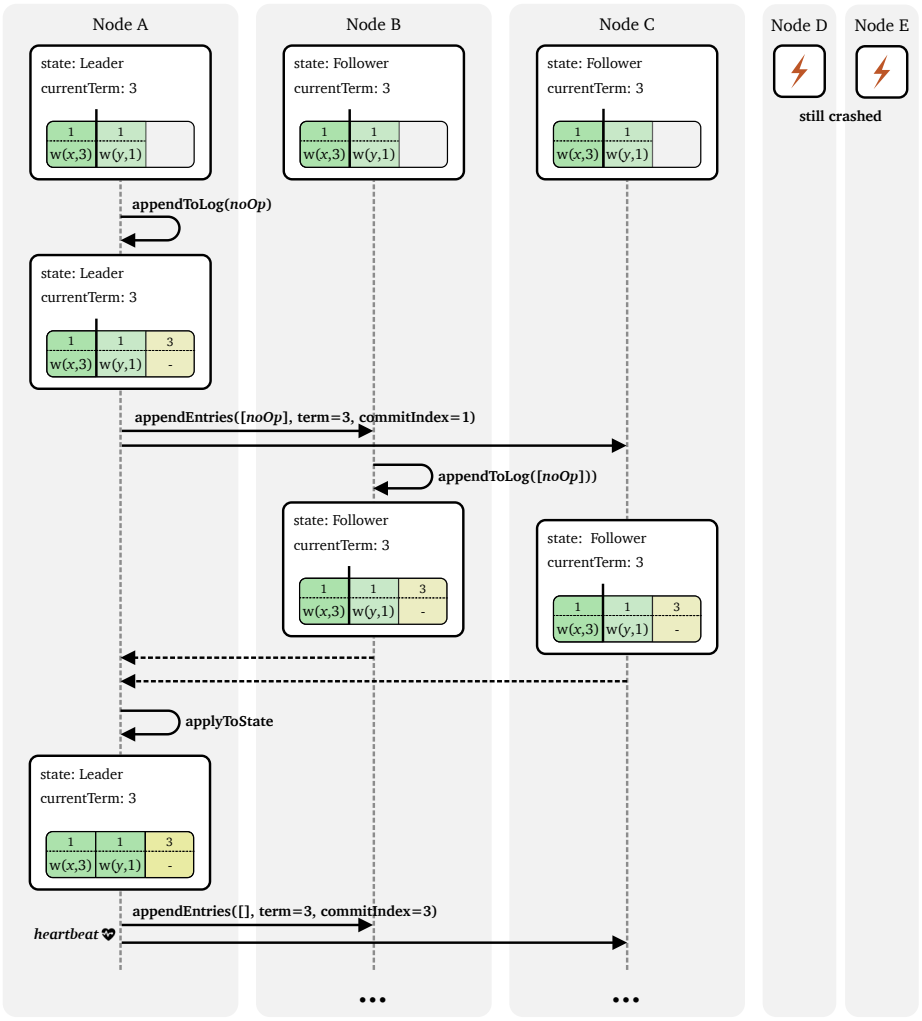


Figure 2.56: After a successful leader election, the remaining uncommitted log entries are committed. Since the Raft protocol forbids committing entries of an old term, a no-operation is appended and replicated to update all logs to the newest term.

2.2.9 Log Compaction

Raft allows for log compaction by creating snapshots of resulting states from committed prefixes of the log, following what we outlined in subsection 2.1.9.2. For this, Raft introduces another message, called `installSnapshot`. We refer to the extended version of the original paper for details on this message and the compaction protocol [117].

2.2.10 Network Reconfiguration / Cluster Membership Changes

We discussed Raft from the perspective of a fixed cluster configuration. In practice, clusters are often reconfigured during their lifetime. Events that result in a change in cluster membership include upscaling outdated machines, replacing crashed servers, outscaling when adding new partitions, rebalancing when a node has proportionally too much load, or increasing or decreasing the replication factor. In some cases, such as in containerized environments, it may be cheaper and more convenient to discard a crashed node and immediately boot a new node with a state snapshot, rather than rebooting a failed node. Raft supports this conveniently and natively, compared to other replication protocols. We won't explain this in detail here and refer to the extended version of the original paper [117].

2.2.11 Cost of Replication

In theory, maintaining strong consistency is achievable but expensive. Raft is used extensibly in production systems, including those that manage a lot of data with very high throughput. We will show some of those throughout later chapters of this work.

In general, the performance of Raft is similar to other consensus algorithms, such as Multi-Paxos. The theoretical cost of replication of Raft can be measured by the number of replicas. It is based on the network roundtrip and the I/O of the cluster nodes to write the log entries to disk. When we consider the best case as illustrated in figure 2.53—ignoring the case of failures and leader election—we can estimate the additional latency per write by a single additional round-trip from a leader to the half of the nodes with the lowest inter-cluster latency. A very slow node won't affect the overall latency since only a majority of nodes is needed to confirm a write. But, we also need to incorporate additional I/O overhead of the Raft log. In the Raft dissertation, a FIFO buffer is suggested to put in place between the raft log and the replication engine, so multiple threads at the leader can take care of writing to the log and sending `appendEntries` requests in parallel.

There are some extensions to Raft that aim to improve latency for reads, writes or both. They can only reduce latency to a certain degree, since there is a physical limit of disk speed and the speed of light. We will outline them in section 2.2.13.

The challenge of every Raft implementation is to achieve the optimal trade-off between the cost of replication and data access availability. For even more effective latency improvements, the basic idea is to reduce the number of Raft protocol instances, if possible, meaning that raft log entries are sent in batches, reducing

the overall number of network roundtrips dramatically. This is especially useful in wide-area networks, while it poses the risk of data loss once the leader fails. We use a comparable approach in our implementation in section 3.3.7.7.

2.2.12 Client Interaction

Other than most consensus algorithms, the Raft dissertation also covers how clients should interact with the cluster for lowest possible latency. It explains how clients can discover a cluster and its current leader, how to avoid possible duplicate requests du to lost intermediate acknowledgements, and also how to linearize writes in case one drops the leader property for read requests. We want carry this topic further here, but refer to corresponding chapter 6 in the dissertation [88].

2.2.13 Possible Raft Extensions

To conclude this subchapter, this section outlines recently published Raft extensions that tackle different challenges and shortcomings of the original Raft proposal. It should be noted that many of these extensions add further complexity to the protocol, which is contradictory to its initial purpose of providing understandability.

2.2.13.1 Multi-Raft

Multi-Raft is the most important extension of Raft and has been implemented in most production systems. With Multi-Raft, the concept of *Raft groups* is introduced to allow to scale with the cluster load. This comes in particularly handy when partitioning, where each Raft group manages a single partition. Each Raft group contains their own leader and manages their own state machine instance. This helps preventing that a single leader for the whole cluster would become a bottleneck by evenly distributing the write load among nodes. Multi-Raft additionally optimizes heartbeat behavior by allowing a node that is the leader in many Raft groups to send a single `appendEntries` message for all groups, instead of one message per group. There is no dedicated literature that describes Multi-Raft, though. Most of its descriptions come from actual implementations.

2.2.13.2 Asynchronous Batch Processing

Li et al. recently suggested to reduce the latency of Raft with the introduction of asynchronous batch processing in Raft [136]. They introduce a new *proposal* mechanism, allowing clients to asynchronously propose writes, which are then processed in batches in the consensus and replication engine. This works under the

assumption that network failures are rare, relying on retransmission mechanisms of the underlying network transport protocol (for instance, TCP).

2.2.13.3 Byzantine Fault Tolerant Raft

There are certain approaches in research on byzantine fault tolerant versions and derivations of Raft. Since we do not focus on byzantine fault-tolerance, we only list them here for reference. Notable ones include Tangaroa [137], Validation-Based Byzantine Fault Tolerant Raft [138] and this nameless one [139]. Some of them are used in blockchains.

2.2.13.4 Learner Roles

One of the most popular services that implements Raft, etcd (which backs Kubernetes), added the concept of learners, similar to that of Paxos, to Raft [140]. They introduced learners to decrease read latency and to put load of from the leader. With increasing load, the leader is more likely to delay the delivery of heartbeats, and therefore followers are more likely to trigger leader elections when they shouldn't, which can lead to serious problems.

2.2.13.5 Read Scalability

Similar to the concept of learner roles, Arora et al. propose to allo for *quorum reads*, which means allowing followers that succesfully applied the latest committed entries of the leader to answer read requests as well [141].

2.2.13.6 Elasticity, Geo-Distribution and Auto-Scaling

The original Raft protocol does not provide lower latency to users by leveraging geo-replication due to its property of having a single leader at a time. With multi-Raft and partitioning, this can be mitigated to some level. But there is also literature on how to make Raft truly elastic by leveraging its cluster reconfiguration capabilities, supporting auto-scaling and geo-distribution. Xu et al. introduces *Geo-Raft* to extend Multi-Raft with two additional node roles: *secretaries* which takes log processing for the leader and *observers* which process read requests for followers [142].

2.2.13.7 Vertical Scalability

Deyerl et al. propose a Raft-based replication protocol called *Niagara* in which the process of appending new log entries is parallelized across multiple Raft instances [143]. It allows a Raft node to scale vertically with the number of cores and network

cards on the machine.

2.2.13.8 Weakened Consistency Constraints

Wang et al. propose to rethink the consistency requirements when using Raft for key-value stores [144]. The authors claim that some of the constraints are not necessary to have consistency for distributed key-value stores. This is related to other approaches we have discussed for key-value stores, such as chain replication or optimistic replication.

2.3 Event Stores and Temporal Databases

> Database: the information you lose when your memory crashes.
>
> — Dave Barry

Event stores are a type of database optimized for storing and managing immutable event streams. They where developed out of *temporal databases* [145].

Temporal databases have been the subject of research since the early 1980s [146], [147]. Temporal tables for relational databases where first proposed in 2011 with the *SQL:2011* proposal, including *time validity period* tables. Instead of overwriting a table column on an update, those tables will create snapshots of the last column state on each update, annotated with its period of validity, to preserve history.

Distributed databases became important with the advent of NoSQL databases like MongoDB, which—at least in their early years—where often only eventually consistent in order to ensure availability, partition tolerance, and performance. The discussions around distributed databases that drop ACID properties for the sake of performance started in 2009[33]. Around the same time, distributed temporal and time-series databases where discussed. They gained popularity in industries and businesses with the rise of IoT and Big Data and the growing demand for organizations to perform present-day and real-time data processing, as well as event stores with event-driven software architecture paradigms.

Distributed event stores generally manage append-only log structures with immutable events. This shifts the discussion of ACID properties and transactional consistency to a discussion of immutable and ordered sequences of state changes. Relational, but also NoSQL databases, generally manage intermediate and final states of real-world objects, while event stores manages the events that describe

[33] https://web.archive.org/web/20110716174012/http://blog.sym-link.com/2009/05/12/nosql_2009.html

the state changes. Thanks to the append-only property, event stores are in general tremendously faster than relational databases, and reduces the need for transactions when combined with methods such as complex event processing (CEP). Note that this is not always comparable, because to describe the same state as in a relational database, you need to write and read multiple events.

We present a selection of distributed temporal databases, event stores and time-series databases in subchapter 3.2.

2.3.1 What are Events?

An event[34] is a notification about an observation from the real world. An *event stream* is a temporally ordered sequence of events and thus a special form of a data stream. An event belongs to a certain *domain*, which is expressed by the *event schema*. Typically, an event stream contains only events of a single schema [148]. Event streams can have multiple order relations, such as the insertion time and the event time (see section 2.3.2), while supporting multiple orders is generally not trivial in implementation.

We define events and related concepts using the following notation, based on some of the definitions of Koerber [149] and Seidemann et al. [10]):

Definition 2.3.1 (Event)**.** An event e_t is a tuple of several non-temporal, primitive attributes $(a_1, \ldots, a_n)$ that resembles a notification about observations of facts and state changes from the real world at a timestamp t[35]. An event is immutable and persisted in an event stream.

Definition 2.3.2 (Event Stream)**.** An event stream E is a potentially unbounded sequence of events $\langle e_1, e_2, \ldots \rangle$ that is totally ordered by event time. We call the domain of event streams $\mathscr{E}$.

Definition 2.3.3 (Time Window)**.** $E_{[t_i,t_j)}$ refers to a time window of an event stream, which is a slice of the stream containing all the events with timestamps in the time interval $[t_i, t_j)$ (excluding events at timestamp t_j).

A time windowing function W_d applied to an event stream E results in a non-overlapping sequence of such time windows. The time windowing function for the

[34] Not to be confused with the definition of an event in probability theory.

[35] There can be two events at the same timestamp t. In general, the subscript of e denotes the index of the event in an event stream, based on event time ordering. We ignore this case in this work to simplify the following discussions and because it is not needed for understanding. Unless otherwise specified, e_t denotes an event that happened at timestamp t in event time.

duration d is defined as

$$W_d(E) = \langle \\ E_{[t_1, t_1+d)}, \\ E_{[t_1+d, t_1+2d)}, \\ \ldots \rangle$$

Note that we use a simplified definition of windowing here as we only care about tumbling time windows, which means that windows don't overlap.

While *temporal facts* describe real-world facts (intermediate states) that are valid in a given period of time, an event often describes a occurence that leads to a change of such a fact. For example:

Table 2.12: A temporal table for temperature measurements

Sensor	**Temperature**	t_s	t_e
1	34.7	2022-02-13T12:31:42	2022-02-13T12:36:11
1	36.1	2022-02-13T12:36:11	2022-02-13T12:38:52
1	35.2	2022-02-13T12:38:52	2022-02-13T12:41:12

Table 2.13: An event stream for temperature measurements

Sensor	**Temperature**	t
1	34.7	2022-02-13T12:31:42
1	36.1	2022-02-13T12:36:11
1	35.2	2022-02-13T12:38:52

In table 2.12, one measurement of temperature is completed once the next is started, since the end of the validity period is known only with the start of the next measurement. In table 2.13, the same measurements are described by the event immediately at the time of their occurence only. The one table can be derived from the other. In continuous observations, the latest row of the temporal table describes the latest known state of the observed entity. The next example is even more figurative:

Table 2.14: A relational table for a person

Name	**Lives in**	**Born on**	**Works at**
John	Berlin	1985-03-23	ACME Corp

Table 2.15: A temporal table for a person's residence

Person	**Lives in**	t_s	t_e
John	New York	2015-05-01	2017-03-01
John	Ontario	2017-03-01	2020-06-01
John	Berlin	2020-06-01	∞

Table 2.16: An event stream for a person's workplace

person.job.applied		
Person	**City**	t
John	New York	2015-05-01
John	Ontario	2017-03-01
John	Berlin	2020-06-01

While the relational table in 2.14 only shows the latest known state for that person, we can model both `Lives in` and `Works at` using temporal tables or event streams.

2.3.2 Time in Event Stores

Temporal databases often come with multiple axes of time. Common axes are:

- **Event time**: The time the event occured in the original application (also referred to as *application time* or *valid time*),
- **Arrival time**: The time when the event arrived at the event store (also referred to as *transaction time*),
- **Insertion time**: The time when the event was stored and processed in the event store (also referred to as *decision time*).

In most systems, there is no distinction between arrival time and insertion time. On the event time axis, some systems—especially temporal databases—not only contain one timestamp per event, but two to demark the start and end of the *temporal validity* period $[t_s, t_e)$ of this event. In databases storing temporal facts instead of events, this is used to describe the period of time that this temporal fact was valid.

To ensure strong consistency without additional out-of-order handling on queries, the ordering of the events based on arrival time and insertion time must be the same. We will elaborate this further in subchapter 3.3.

High-Resolution Timestamps. In practice, many applications and sensors provide a very low time resolution of milliseconds, microseconds or sometimes even nanoseconds (such as in experimental setups of particle physics). In addition, consecutive events may carry equal timestamps. The ordering of events is therefore at least non-decreasing. Event stores need to support this property without violating consistency. Due to multiple sources of events with slightly offset clocks and especially with very high throughput, it is a challenge to synchronize event time. General computer operating system are not able to resolve event time at submicrosecond level, especially as CPU multi-tasking/scheduling is not able to provide this resolution. This requires solutions including optimizations on the network level to have such high-resolution timestamps, such as the *Precision Time Protocol* (PTP) [150].

2.3.3 Differentation of Event Stores and Time Series Databases

An event store contains and manages events in multiple *event streams*, while a time series database stores and manages *time series*.

A time series consists of multiple *data points* on a time axis. A data point is a record of a *measurement* of an entity. Usually, data points on time series are uniformally distributed on the time axis, in contrast to events, that usually occur in unpredictable patterns [151].

Time series databases support both temporal correlation and compression, but typically do not provide query support for operations such as pattern matching and complex event processing.

Table 2.17: A time series for temperature measurements

t	**measurement**	**field**	**value**
2022-02-13T12:31:42	temperature	sensor:1	34.7
2022-02-13T12:32:41	temperature	sensor:2	32.7
2022-02-13T12:32:41	temperature	sensor:1	36.1
2022-02-13T12:33:392	temperature	sensor:1	34.9

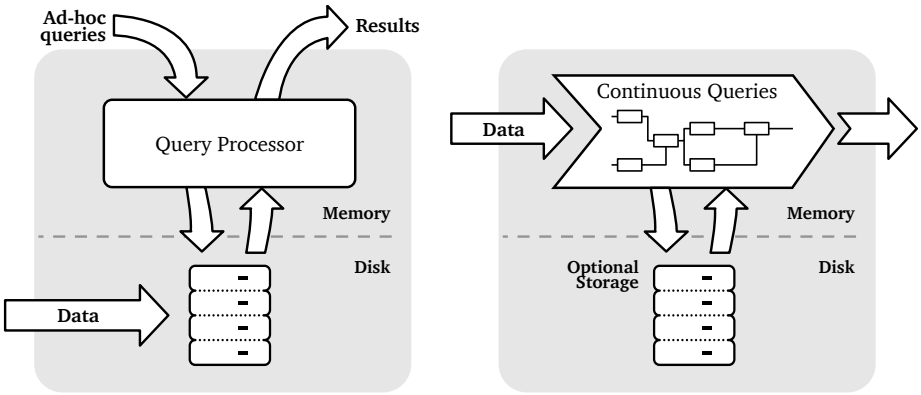
In table 2.17, a typical time series table is shown. It represents the same observations as the event stream in table 2.13, but its structure is different: while an event stream contains ordered events matching a single schema per stream with any arbitrary payload, a time series row represents a measurement at a specific time. A measurement has a specific type, a field of the measurement and the actual measurement value for that field. Besides that, it can contain any number of additional labels or tags (which are further columns). As a result, time series typically contain a single measurement per data point, and multiple different measurements per time series, while an event can contain multiple measurements that happen at the time of the event. A time series measuring both temperature and humidity would be represented by two event streams, one for each measurement, since changes in temperature and humidity may not happen at the same time. While a measurement is triggered by a measuring device, so generally producing equidistant data points, independently of an actual change of the subject of measurement, an event is triggered by the change of the subject itself, which could occur at random intervals. This also means that in time series, the measuring system is responsible for generating the data point, and in event streams, the system where the event occured produces the event. Both time series and event data are immutable. We listed all the differences and similarities in table 2.18.

Table 2.18: Differences between Event Stores and Time-Series Databases

	Event Stores	**Time Series DBs**
Store Format	Event Stream	Time Series
Granularity	Events	Measurements
Record distribution	Random	Uniform
Mutability	Immutable	Immutable
Record producer	Measured system	Measuring system
Scope	One schema per stream	Multiple measurement types per series
Record Payload	Arbitrary	One measurement + labels

2.3.4 Relationship to Data Stream Management Systems

A *data stream management system* (DSMS) extends a database management system by managing continuous streams of data, and *continuous queries* on those streams [152]. This means that instead of storing data first and then allowing systems to run ad-hoc queries, data directly runs in streams through a stream processor, where continuos queries are executed on the streams in real-time. Persisting the stream data is optional in such systems, and can occur either in embedded storage or an external database. The data must not neccessarily be events, it can be any kind of ordered data stream, but it appears naturally to be events in many cases. Unlike traditional data management systems, and due to the nature of stream applications, DSMSs must respond to incoming data and deliver results to consumers frequently and almost immediately. Managing unbounded streams with limited memory and CPU is one of the biggest challenges for a data stream management system. A DSMS is not restricted to continuous queries, it can also allow for ad-hoc queries by storing streams permanently. We illustrated the difference between a DSMS and traditional DBMS in figure 2.57, based on [152].



a) Traditional Storage Architecture b) Stream Management Architecture

Figure 2.57: Data stream management system vs. traditional storage architecture. a) In traditional storage architectures, data is directly persisted (often with transaction handling), before it is available for ad-hoc queries. b) In data stream management systems, data comes in as continuous and unbounded streams, and is immediately processed by a continuous query processor. Afterwards or in parallel, it is optionally persisted in the storage.

Data appearing out-of-order must also be taken into account by a DSMS and may be tolerated in controlled bounds. DSMS have a broader and generic scope, and often consist of multiple systems and services. For instance, they could cover complex event processing (CEP) systems: a DSMS allows for any arbitrary real-time data processing, while CEP systems are especially build for event (pattern) detection. An event store that supports continuous queries in real-time can can be considered as a part of a DSMS, responsible for managing, serving and persisting continuous streams.

2.3.5 Use Cases and Challenges

Event stores, DSMS and related database systems support a wide range of use cases. For example, such systems allow to measure and track software and hardware metrics, including

- System health monitoring,
- Network analysis,
- Fraud detection,
- As well as usage and behavioral data analysis.

These systems are also used in measuring sensor data and events, including

- Personal health data (e.g., from fitness trackers),

- Medical data,
- Physics and biology experiments,
- Video data (e.g., from surveillance cameras),
- IoT,
- Industry 4.0,
- Smart grids,
- And smart cities.

From the system design perspective, there are the following applications:

- Event-sourcing and event-driven systems,
- Message brokers,
- Complex event processing (CEP),
- Stream processing,
- Process mining,
- And machine learning.

Many of this cases require fault-tolerance, high availabilty, very high write throughput and low latency for queries, while scaling with huge numbers of sensors and devices. This creates a strong need for distributed event stores. We will encounter some of these use cases again throughout this work.

2.3.5.1 Replicated Event Stores as a Replication Service

Similar to ZooKeeper, replicated event stores can be deployed as a replication service themselves, running next to a distributed system and offering the service to store and replicate the commands of this system. From this point of view, an event store turns out to be a replicated log—with some powerful indexing features. Event-sourced systems actually rely on the event store to hold the (replicated) source of truth for the overall system and individual application state.

However, this approach is not recommend for all use cases, as it adds additional unwanted latency by enforcing back-and-forth between the services, which is also a reason why Kafka decided to replace ZooKeeper by a quorum-based, embedded replication protocol.

2.4 ChronicleDB, a High-Performance Event Store

In this chapter, we introduce the particular event store that this work focuses on: *ChronicleDB, a high-performance event store* [10]. Throughout subsequent chapters of this work, we will implement a replication layer that allows us to run ChronicleDB as a distributed event store in both the cloud and on the edge, while maintaining the capabilities of standalone operation and embedding in edge devices such as sensors or mobile devices. In this introduction, we will not explain ChronicleDB in detail, but focus on the core aspects of ChronicleDB that are relevant to operating in a replicated environment. For further details and the entire specification, we refer to the work of Seidemann et al. [10] and Glombiewski et al. [153], and also to the dissertation of Körber on online and offline stream processing [149].

ChronicleDB is an event store optimized for a high write performance under fluctuating data rates, but it also comes with powerful indexing capabilities to support a variety of continuous and ad-hoc queries. Experimental evaluation has shown that ChronicleDB outperforms competitors in terms of read and write performance [10]. Before this work, ChronicleDB was designed to operate in two modes: either as a serverless library to be embedded in another application or as a standalone database server. It also provides fault-tolerance and fast failover behavior in embedded and standalone operation, which comes in handy as it increases overall availability and reliability also in distributed operations.

2.4.1 Use Cases

ChronicleDB serves a broad range of use cases, due to its support for various query types. It supports both continuous and ad-hoc queries, allowing it to be used in both real-time and batch processing systems.

ChronicleDB is therefore suited for all applications that rely on low-latency processing of events that may occur in random, but very short intervals. We listed these use cases in section 2.3.5.

When discussing these use cases, it is important to seperate between those with a demand for strict ordering guarantees, requiring strongly consistent streams at read time, and those that are ok with lower safety guarantees in exchange for lower latency and higher throughput. It is also important to look both at the event producer and consumer side in such an argument. We conduct this argument in detail in section 3.3.4.

Relationship to DSMS. ChronicleDB may not be considered a DSMS in the classic sense, but in an extended sense: although the stream data is stored first on disk, it is also immediately available for continuous queries, and running aggregates are stored and continuously updated. In addition, ChronicleDB extends the requirements for a DSMS, as on-the-fly processing of event streams without persisting the streams is not always a sufficient solution. In many use cases, it is necessary to keep the collected data permanently, e.g. for future ad-hoc queries, to compare real-time processing results with historical data, and to detect trends and changes.

For instance, in the realm of IT security, historical data is crucial since critical security incidents must be reproduced to derive new security patterns. For this purpose, various streams arriving at a high and fluctuating rate must be stored in a persistent storage [10].

ChronicleDB was designed not only for performance, but also to support this cases, since at the moment, there is a lack of systems supporting high throughput writes, continuous queries, ad-hoc temporal queries, and fault-tolerance at the same time.

2.4.2 Relationship to Event Stream Processing.

ChronicleDB is designed to run next to event stream processing systems. ChronicleDB offers outstanding support for sliding window aggregation and pattern matching queries. Thanks to its lightweight indexing, it can compute these aggregates incrementally as new events arrive. Secondary information is kept within the primary index, which allows to avoid random writes during index creation. This also removes the need to access multiple indexes when querying multiple attributes, speeding up both real-time and ad-hoc queries. This makes ChronicleDB a great companion to event stream processing systems. As mentioned before, ChronicleDB does not only support real-time queries, but it also provides the ability to replay continuous queries again on historical event data.

2.4.3 Requirements and Architecture

> The log is the database.
> — Seidemann et al.

In ChronicleDB, the major design principle is: *the log is the database*. ChronicleDB leverages the properties of append-only log structures and in addition, it tries to avoid additional logging, as the considered append-only scenario does not cause

costly random I/Os, and therefore it does not incur buffering strategies with no-force writes. There is one exception in case of out-of-order arrival of events: these are written into a write-ahead log first, since events should be kept ordered by event time.

In figure 2.58, we have outlined the architecture of ChronicleDB based on the descriptions in the original paper. ChronicleDB consists of a query engine to serve both continuous queries and ad-hoc queries, either via or a Java API, HTTP API or ChronicleDB-flavored SQL. The same accounts to writes. Reads and writes are served via load scheduler (illustrated in figure 2.59) from and to the storage engine. The storage engine provides powerful indexing capabilities (section 2.4.4), a block storage layout with very fast lossless compression (section 2.4.3.3), and event serialization.

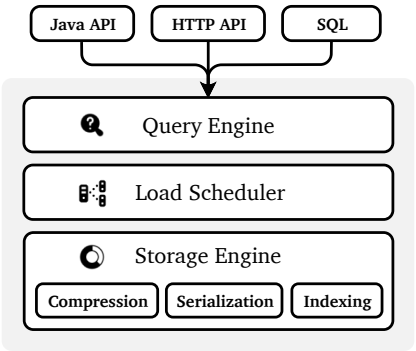


Figure 2.58: Overview of the ChronicleDB architecture

2.4.3.1 Events and Streams

ChronicleDB supports temporal-relational events (see definition 2.3.1). Event streams in ChronicleDB are append-only log structures similar to multi-variate time series in time-series databases, but with non-equidistant timestamps. Timestamps can either refer to insertion time or event time (see subsection 2.3.2). Since event time is more meaningful in queries for most consumers, ChronicleDB tries to maintain a physical order on application time [10]. Once an event is inserted into a stream, it is immutable.

Inserting events in standalone ChronicleDB happens with multi-threading support to increase write throughput when multiple streams are served at the same time. Multiple job workers are instantiated to care of writing to the right stream and the right disk on the operating machine. This is illustrated in figure 2.59.

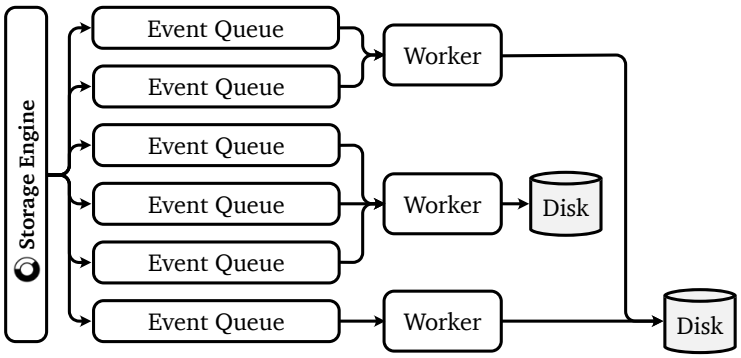


Figure 2.59: Illustrative example of a ChronicleDB queue topology for 6 event streams

2.4.3.2 Queries

Besides continuous queries, ChronicleDB has special support for *time travel queries* and *temporal aggregation queries*. Time travel queries allow to fetch events for specific ranges in time, e.g. all orders that have been fulfilled in an e-commerce system within the last hour. It allows consumers to derive a snapshot of the state of the depicted real-world at any point in time. Temporal aggregation queries give a comprehensive overview of the data, e.g., the average number of fulfilled orders for each day of the week during the last three months. Moreover, ChronicleDB supports queries on *non-temporal attributes*, e.g., all fulfilled orders within the last day for a specific product. It also supports efficient replay of common complex real-time event processing queries, such as pattern matching, at any time.

You can read more on ChronicleDBs query capabilities, such as temporal pattern matching, sliding window aggregation, and replay of continuous queries, in the original paper [10].

2.4.3.3 Storage Engine

To decouple event producers from indexing and disk writing, memory-based FIFO event queues are placed in between producers and disks. Job workers continuously fetch entries from these queues and write them to disk, while adding the index entries. This is shown in figure 2.59. This also allows us to run ChronicleDB efficiently in a distributed setting, since we do not need to wait for entries to be persisted on disk before acknowledging them to the replication layer.

The storage layout of ChronicleDB is partitioned into macro blocks. A macro block consists of C-blocks (compressed block), which in turn consist of compressed L-blocks.

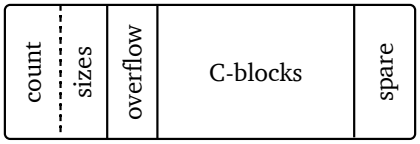


Figure 2.60: Layout of macro blocks in the ChronicleDB storage

L-blocks (logical blocks) are of the size of a physical disk block. Macro blocks are of a fixed size with a variable amount of C-blocks and provide the smallest granularity for physical writes to disk. This setup allows for improved fail-over behavior. We refer to the original paper to learn more about ChronicleDB's compression and other storage layout features, as this is beyond the scope of this work [10].

The macro blocks also contain a *spare space* that reserves space for the insertion of out-of-order events. While the macro block is generally dense, this spare space allows for efficient insertion of out-of-order events since it helps to avoid costly block overflows.

The overall block design supports very economic storage utilization that allows to store data for the long term, while it allows for high-throughput writes at the same time. ChronicleDB runs efficienlty on centralized storage system even with cheap disks, such as in embedded systems like sensors and IoT devices.

2.4.4 TAB$^+$ Tree Index

The index of Chronicle is built on a *temporal aggregated* B^+ (TAB$^+$) tree. The TAB$^+$ tree is a novel approach, based on a B$^+$ tree that uses the timestamp of an event as its key. This approach delivers lightweight indexing that also allows small sets of aggregates to be maintained at the attribute level, such as `min`, `max`, `count` and `sum`. The aggregates are calculated and stored on the various levels of the tree, allowing to update the aggregates on the go with each arriving event. As mentioned before, secondary data is kept within the primary index, allowing to run faster queries and omitting further random writes during index creation. The index leverages the property of *temporal correlation*, which describes the observation that attributes of events occurring within a small time interval are often very similar. We illustrated the TAB$^+$ tree in figure 2.61. For a detailed explanation of ChronicleDB's indexing approach, including how the tree is built, how it refers to the storage layout, how it allows for fast crash recovery, how secondary indexes works and more, we refer here to the original paper [10].

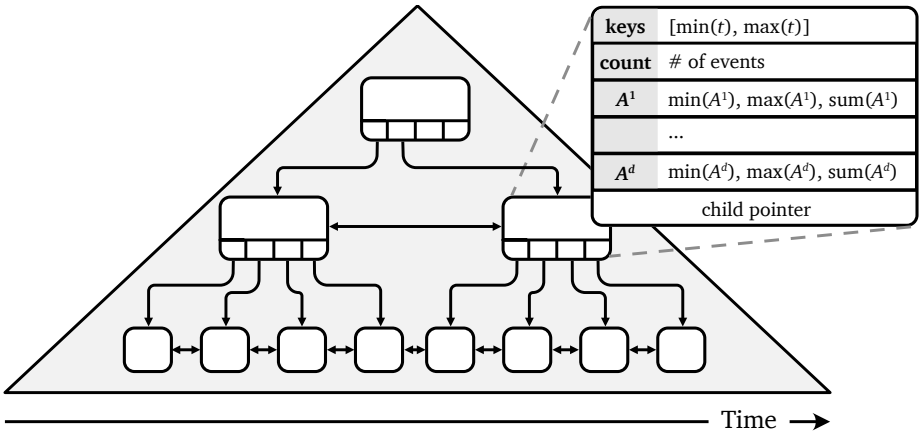


Figure 2.61: The TAB$^+$ index layout. The nodes and leaves of the trees are connected in both directions, supporting faster queries. Each node contains pointers to its children, the minimum and maximum timestamp of the events in the branch as the index keys, the number of events per branch, and the basic aggregates per property defined in the event schema.

2.4.4.1 Time-Splitting

ChronicleDB allows users to align the organization of the data to their most common query patterns by introducing *regular time splits*. On a per-stream level, the user can decide the maximum period of time that one time split should cover. Once a time split reaches this size, the split is closed and a new TAB$^+$ tree instance is created for the subsequent current split in a seperate file. This makes historical aggregation queries on predefined time ranges more efficient, such as the total sum of fulfilled orders on a weekly basis, but it also increases write performance.

In rare cases, historic event data should be deleted (e.g., to comply with GDPR). ChronicleDB supports the removal of such events at the granularity of time splits.

2.4.5 Summary

The ChronicleDB event store is a novel approach that allows for efficient online stream processing, as well as long-term persistence of events with high throughput that outperforms other temporal database systems, while supporting ad-hoc queries and reliable and fast replay of continuous queries. It runs embedded in other applications on low-cost hardware, such as IoT sensors, as well as a standalone server. Hence, it aims to serve a broad range of use cases, which me must take into account when designing a replication layer for ChronicleDB in chapter 3.

Chapter 3

Protocol Decision and Implementation

In the previous chapter, we introduced the reader to the realm of replication, explained the various properties of a replicated distributed system such as reliability, fault tolerance, consistency, and latency, and articulated a rationale for why a distributed database should be run with replicated data. We then discussed various consistency levels and replication protocols and provided guidance on how to decide on the appropriate model and protocol. We then looked at event stores and the requirements for them and introduced ChronicleDB, the event store we want to move to a distributed setting.

However, we have not yet provided explicit guidance on which consistency model and replication protocol is best suited for ChronicleDB under specific circumstances and use cases. Therefore, in this chapter we will demonstrate such guidance in three steps:

- First, we look at existing implementations of replication protocols for data-intensive applications and how they differ, focusing on event stores and other systems comparable to ChronicleDB.
- Second, we will take a closer look at the requirements of a potentially distributed ChronicleDB and decide on a consistency model and replication protocol, pointing out the trade-offs and how to mitigate them.
- Finally, we will describe the system design and architecture of a solution and discuss our demo implementation of such a replicated event store: ChronicleDB on a Raft.

In the remaining chapters, we will then evaluate our implementation, compare

the trade-offs with those of popular and comparable distributed database systems, and conclude by presenting the key learnings of this evaluation and pointing to possible future work.

3.1 Methodology

> To think, you have to write. If you're thinking without writing, you only think you're thinking.
>
> — Leslie Lamport

The main goal of this thesis is to identify a consistency model and replication protocol suitable for an event store with a specific set of requirements, namely ChronicleDB. This includes a detailed discussion of the advantages and disadvantages of the models in question. This section outlines the methods used to achieve this goal. A huge portion of this discussion already happened the chapter 2, such as in the subsection 2.1.8, where we layed out the criteria to consider when deciding for a consistency model.

The Research Approach. Throughout this work, we use an exploratory research design. So far, we have explored the realm of distributed systems and replication. In the remainder of this work, we will conduct a qualitative comparative analysis of replication protocol implementations. We also gather the requirements for a distributed version of ChronicleDB and the tolerable range of trade-offs. Based on this assessment, we design a distributed version of ChronicleDB with replication in order to implement it as a prototype.

We use benchmarking as a quantitative method to analyze our prototype implementation and to either confirm or refute our predicted results. In this benchmarking, we compare our implementation with ChronicleDB in standalone mode running om a single node and compare the results with findings in the literature, with respect to other distributed databases, to determine if the performance trade-off we experienced is within the expected range.

Significance of our Results. The results of our qualitative research are significant since we covered a broad range of relevant literature in our analysis. We analyzed both the history and present of replication in distributed systems for fault-tolerance, dependability, consistency and scalability. We examined not only industry standards, but also not-so-popular approaches that have received less attention in the academic

environment.

Our quantitative results are (a) reliable and (b) valid because (a) our benchmarking setup can be easily replicated (see the Appendix for a Jupyter notebook that can be run against our implementation of ChronicleDB) and (b) our results clearly depict the metrics needed to measure trade-offs and overall performance. We also point out to the limitations of our evaluation and possible future steps.

3.2 Previous Implementations

> If I have seen further it is by standing on the shoulders of Giants.
>
> — Isaac Newton

In subchapter 2.1, we discussed consistency models and several replication protocols. In this subchapter, we take a look at existing implementations of replication protocols in practice by reviewing several selected distributed databases for event-like data. We will investigate their approaches to replication, the impact of these approaches on performance and client consistency, and what we can learn from them. It is important to evaluate the impact of replication protocols in practice, as some theoretical constraints have a negligible impact on practical behavior, while the impact of some other considerations is evident in practical applications.

3.2.1 Criteria for Review and Comparison

Since there are a multitude of distributed databases, we need to narrow the scope for our consideration.

We are looking for databases and similar system that store and manage event-like data, such as time series or messages. Our focus is on replication mainly for fault-tolerance and increased availability. We do not look at databases that only offer geo-replication. We also do not look at decentralized systems, such as blockchains.

3.2.2 Distributed Event Stores and Time-Series Databases

We look at various event stores, time-series databases and related applications, such as pub/sub messaging systems, and summarize and compare their properties in table 3.1.

Table 3.1: The list of event stores and similar systems studied in this work, and their primary dependability and consistency properties

System	Consistency Model	Replication Protocol
	Event Stores	
EventStoreDB	Strong—Eventual (?)	Custom
Azure Event Hubs	Strong—Eventual (per partition)	Unknown (proprietary)
	Time-Series DBs	
InfluxDB	Meta: Strong. Data: Eventual	Meta: Raft. Data: Custom
Apache IoTDB	Strong	Raft (Ratis)
	Acounting Databases	
TigerBeetle	Strong	Viewstamped Replication
	Pub/Sub & Message Brokers	
Apache Kafka	Meta: Strong. Data: Strong—Eventual (per topic)	Meta: ZooKeeper (Now), KRaft (Proposal). Data: Primary-Copy
RabbitMQ	Strong-Eventual (per queue)	Raft-based

3.2.2.1 InfluxDB

The original ChronicleDB paper mentions InfluxDB as the "most related storage system". InfluxDB is an open-source time-series database that was natively developed to operate in a distributed cluster. It is not optimized to run on a single machine or embedded in another application, such as ChronicleDB. As a time-series database, it works best with univariate time series data and does not support queries for complex event processing. It provides its own storage layout, called time-structured merge tree (TSM) (which, as the name suggests, is somehow based on *log-structured merge trees* (LSM)). It is also suitable for long-term data storage.

InfluxDB does not provide strong consistency for data. They provide two seperate

layers of consistency[1] [154]:

Meta Nodes. Meta nodes store metadata for cluster management with strong consistency using a distinct Raft subsystem. Meta nodes cover the following data:

- Cluster membership,
- Retention policies,
- Users,
- Continuous queries,
- Shard metadata.

Data Nodes. Data nodes store the actual payload (the time-series data itself, the indexes and schemas) seperately and with an own replication scheme that provides eventual consistency (in practice only under very high throughputs, according to an own statement), called *hinted handoff*. Therefore, the data nodes can already be operated with a low replication factor of 2 (without a quorum), while a quorum mode is also possible - but still only eventually consistent.

In the past (before version 0.10), InfluxDB also used Raft for data nodes[2] [155]. Since they were not able to meet throughput requirements with their implementation of Raft since "In practice it was inefficient and complex to implement correctly", they decided to remove Raft for data nodes and to introduce an eventually consistent protocol. Their latest statement to this decision reads: "InfluxDB prioritizes read and write requests over strong consistency. InfluxDB returns results when a query is executed. Any transactions that affect the queried data are processed subsequently to ensure that data is eventually consistent. Therefore, if the ingest rate is high (multiple writes per ms), query results may not include the most recent data" [156].

Geo-Replication. While they do not provide their own solution to geo-replication and disaster recovery, they point out to possible solutions using other tools like Kafka or specialized tools for InfluxDB built by third parties[3] [157].

[1] Also see their argumentation why they are neither CP or AP, according the the CAP theorem https://www.influxdata.com/blog/influxdb-clustering-design-neither-strictly-cp-or-ap/

[2] The documentation on old versions with full Raft, but in alpha state, can still be found in their archives: https://archive.docs.influxdata.com/influxdb/v0.9/.

[3] Also see for reference https://github.com/influxdata/influxdb-relay and https://github.com/toni-moreno/syncflux.

3.2.2.2 Apache IoTDB

Apache IoTDB (Database for Internet of Things) is a time-series database specifically made to work in an edge-cloud setting. It can be deployed on all three layers in edge computing: embedded on edge appliance, e.g. on IoT devices, in the terminal layer, standalone on the edge layer and distributed on the cloud [158]. This is actually pretty similar to our approach for ChronicleDB. The open-source project started in 2019 and was adopted by Apache after it was accepted as an Apache Incubator project.

Following is a list of IoTDBs key features:

- Apache IoTDB is developed to work closely with existing open-source stream and batch processing tools, providing out of the box connectors for Apache Spark, Flink and Hive (Hadoop)[4].
- In IoTDB, data is partitioned by time splits (here, it is called *time slices*).
- It comes with a custom storage layout.
- It features a config node and a data node group, similar to InfluxDB.
- In addition, data and schemas are partitioned seperately, while the replication factor and consensus protocol for both can differ.
- Its default replication protocol is Raft.
- To move data between the edge-cloud layers, if provides an own sync protocol and service, that replicates the time-series data on a file level.

The project is still under heavy construction, especially its consensus layer. Currently, they use *Apache Ratis* to implement the Raft protocol—to say it in advance: We have also opted for this library. At the moment, they build an aditional adapter layer around it, allowing them to be agnostic of the underlying protocol and framework. This adapter also allows to switch between embedded/standalone and cluster operation modes, without having to build and maintain 2 different architectures[5].

3.2.2.3 EventStoreDB

EventStoreDB is, as the name suggests, an event store, but not a multi-purpose one. The authors of EventStoreDB call it “The stream database built for Event Sourcing”. Event sourcing is in fact its main use case: it is built to store consecutive changes of a state as separate events. While sensor data could also be tracked with EventStoreDB,

[4] https://iotdb.apache.org/

[5] Cf. the consensus module/adapter at https://github.com/apache/iotdb/tree/master/consensus and this recently merged pull request to include a multi-leader protocol (similar to ROWA, cf. subsection 2.1.9.3) behind this adapter https://github.com/apache/iotdb/pull/5939

it is not its main purpose, and it only serves low throughput compared to ChronicleDB and other systems we investigated here: over 15.000 writes/s according to their website [159].

EventStoreDB can be operated in different modes. We haven't found the actual protocol and consistency details and can only make assumptions about the consistency of a EventStoreDB cluster by looking at its various operation modes and the guarantees given in each mode. They provide guarantees on writes, since by default, every write to the cluster needs to be acknowledged by all cluster members, not just a quorum. While this condition can be changed by users, the documentation strongly advises against doing so [160]. They also provide roles for nodes that are similar to consensus protocols: a node can either be a leader, a follower, or a read-only replica (that can be queried, but does not partake in a quorum to acknowledge a write). Per default, the cluster runs in a single strong leader mode, which in combination with a quorum should guarantee strong consistency. They also allow users to opt out of strong leader guarantees, so together with lower write acknowledgement requirements, this may result in eventual consistency.

3.2.2.4 TigerBeetle

TigerBeetle is an append-only accounting database with strict serializability [161]. Since it is used in financial use cases with high business criticality, strong consistency is a must. It uses Viewstamped Replication under the hood, which belongs to the class of state machine replication protocols (see subsection 2.1.10.3). On their website, they claim that they provide a high throughput of approximately 1 million journal entries per second, while providing "extreme" fault-tolerance [162]. They make this a clear unique selling point, since their website is called "TigerBeetle - A Million Transactions Per Second".

The database itself is available open-source, written in the language `Zig`, while they offer a paid premium service.

3.2.2.5 QuestDB

QuestDB is a standalone time-series database. The authors claim that "QuestDB is the fastest open source time series database" [163]. This claim is underpinned by a benchmark suite that is public and can be run by developers themselves, and their benchmark is impressive: on the same setup, QuestDB shows a peak throughput of almost 1 mio rows/s, while InfluxDB only comes up with around 330k rows/s [164].

QuestDB is currently not a distributed database, which may explain the high

throughput. While they provide a cloud offering, this is not fault-tolerant. Based on their roadmap, they aim to support distributed reads and writes in upcoming releases[6], including handling of out-of-order events.

3.2.2.6 Apache Kafka

Apache Kafka is a distributed event streaming platform that can be used in a *publisher-subscriber* (pub/sub) scheme. The pub/sub scheme describes systems that allow publishers to produce a single message or event that is consumes by multiple subscribers [165]. Kafka was originally developed at LinkedIn to track user interaction events on their social media platform [166], and quickly became something of a de facto standard for a variety of industries and companies, including manufacturing and finance. Nowaydays, it supports not only pub/sub message distribution, but also resilient stream processing. Kafka can also be used as a permanent storage for streams, served from a fault-tolerant high-availability cluster.

At the heart of Kafka is a replicated transaction log that manages *topics*. A topic in Kafka is similar to what we call an event stream in ChronicleDB, or what is generally called *channel* in event processing [167]. A topic is a domain of interest that a consumer subscribes to, to receive all messages regarding this specific topic. Topics are partitioned and distributed across nodes, called *brokers*. Each partition is an append only log with strong insertion ordering guarantees. Other than streams, a topic can be configured to operate in multiple ways, such as in a *exactly-once* mode, which guarantees that a message is always delivered, and exactly once, to one or more consumers, even in case of producer or consumer failures. Kafka also allows for atomic transactions across multiple topics.

To allow for such guarantees, the replication layer of Kafka provides multiple levels of consistency for the user to select from [168]. The meta (cluster and topic information) and data layer in Kafka are distributed with different consistency guarantees and replication protocols. Data itself not replicated using consensus, but with some approach similar to primary-copy log replication. Users can decide on the replication and the acknowledgement level per stream. A quorum is not required. Depending on the acknowledgement level, the protocol uses a high-water mark to identify the portion of the log that is already acknowledged [169]. Since Kafka does not differentiate in ordering between insertion and event time, it can operate directly on the topic streams.

Metadata is distributed with consensus, thus providing strong consistency. Cur-

[6] Cf. https://github.com/orgs/questdb/projects/1/views/5 for reads and https://github.com/questdb/roadmap/issues/12 for writes

rently, the metadata replication layer and the leader election of Kafka is built on Apache ZooKeeper [90]. The metadata also includes crucial information for read consistency, such as the topic *offset*, which marks the last consumed event of the topics stream. We introduced ZooKeeper and the ZooKeeper Atomic Broadcast Protocol in subsection 2.1.10.2.

KIP-500: Replace ZooKeeper with a Self-Managed Metadata Quorum (Raft) Since ZooKeeper is an external service and not embedded natively in the distributed data storage system it replicates, it comes with some caveats, especially regarding performance. Therefore, the Apache Kafka team dediced to discontinue the usage of ZooKeeper and to implement their own replication layer for metadata, embedded in Kafka. You can find the motivation to do so in their often-cited *Kafka Improvement Proposal* (KIP) 500 [170]. This replication layer is based on Raft and called *KRaft*, where the *K* stands for Kafka. This is still under development and therefore currently not recommended to be used in production.

3.2.2.7 RabbitMQ

RabbitMQ is a *message broker* built on the *AMQP* protocol. It provides similarities to the *pub/sub* scheme as in Apache Kafka, but it works quite differently. It's main use case is messaging, and it provides powerful routing capabilities for messages that goes beyond subscribing to a topic. RabbitMQ is commonly used in telecom industries thanks to this routing capabilities and because it built upon the *Open Telecom Platform* (OTP) which is a core part of the open-source distribution of `Erlang`.

Most of the distribution and replication design of RabbitMQ is based on the OTP. But recently, the authors introduced Raft for *quorum queues* [171]. This allows RabbitMQ users to opt-in to strong consistency on a queue level. A queue is the equivalent to a channel in event processing [167] or an event store stream. On the RabbitMQ documentation, they highlight that quorum queues should be used in critical cases where fault tolerance and data safety is more important then latency and advanced queue features—effectively saying that a CAP trade-off needs to be made.

3.2.2.8 Azure Event Hubs

Azure Event Hubs is a *Platform as a Service* (PaaS) for event streaming. It provides full integration into the Kafka ecosystem, and allows to natively connect realtime

stream processing and batch processing systems of the Azure service portfolio, as well as event sinks for long-term storage. Microsoft claims that it can ingest millions of events per second [172], since it allows to scale linearly with stream sharding.

Event Hubs allow users to choose the consistency level on a per-partition (shard) basis. Users can choose between eventual and strong consistency. With strong consistency, the ordering of the events are guaranteed. Across partitions, they do not provide consistency, since each partition is covered by its own replication protocol instance [173]. They also do not provide an query consistency layer that restores consistency on reads, so users are adviced to either use single-partition streams or to handle this on the event consumer side.

3.2.2.9 Comparison Results

Our analysis shows that high throughput can be guaranteed even with strong consistency (e.g., TigerBeetle, IoTDB). Most vendors separate data and metadata when it comes to replication. Some vendors point out the tradeoffs and have chosen not to provide strong consistency at the data level, while others let the user decide the level of consistency. What most have in common is that they provide strong consistency for metadata (cluster information, stream watermarks, and other information that is not pure payload data).

3.3 System Design and Architecture

In this section, we present the design of the system, justifying the choices for dependability properties, consistency, replication protocol, libraries used, and implementation steps throughout. We also show what trade-offs we accept and when, and what this means for the final solution. We will start with a theoretical discussion, outlining the system design, and continue with a more detailed presentation. Finally, we present the implementation of our prototype, which we use to perform the benchmarking and examine the overall impact of the chosen protocol on ChronicleDB.

3.3.1 Challenges

> Problems are not stop signs, they are guidelines.
> — Robert H. Schuller

The biggest challenge in this work was to find an appropriate trade-off between latency, availability and consistency, and therefore to find an appropriate consistency model and based on this, finding a replication protocol that satisfies the traded-off requirements. In addition, we tried to find proper ways to soften the consequences of the compromises. The trade-offs could only be made after an investigation of the potentially targeted use cases of a distributed ChronicleDB and a general discussion of consistency in event stores (subsection 3.3.4). In addition, handling of out-of-order events is challenging, especially in the face of consistency. It violates consistency, but not from the point of view of the event store, but rather from the point of view of client applications that expect a reliable stream of events. Some replication protocols require a write-ahead-log (WAL), which violates the "The log is the database" philosophy of ChronicleDB.

Throughout the next sections, we will show how we tackled these challenges, and in chapter 5, we will show how these could be further mitigated in the future. But first, we limit the scope of this work to the part of the challenges that we tried to solve in the following subsection.

3.3.2 Limitations

Due to the complexity of this topic, the research and implementation in this work is limited to a few well-justified considerations.

Geo-Replication. We limit our work to intra-cluster replication. See the conclusion for some hints for further investigations.

Byzantine-Fault Tolerance. We haven't implemented or designed a byzantine-fault tolerant version of ChronicleDB, as this work is limited to a problem scope where byzantine faults are rare (but not yet excluded).

Transaction Handling. So far, ChronicleDB does not support transactions, which is actually rare for event stores in general. This is good for us, because we don't need to think about linearizing transactions and making them atomic, which also increases the latency of such a system tremendously. If transactions are ever needed, especially across partitions, we refer to subsections 2.1.5 and 2.1.9.7.

Distributed Queries. The following implementation in this work focuses on write replication. To query an event stream with shards on different nodes, distributed queries must be performed, which is an entirely separate, complex area of interest that exceeds the scope of this work. We refer here to the literature of distributed queries and implementations in popular distributed database systems [174]–[176]. In this work, we have not yet implemented the query interface, even for single-shard queries.

Multi-Threaded Event Queue Workers and Partitioning. We don't know how the decoupling of local event queue threads by introducing partitioning affects the possible performance that can be achieved with multiple streams through ChronicleDBs own event queue and worker architecture, as with partitioning, the relation is no longer one-queue-per-stream, but one-partition-per-stream. As we move the actual event store implementation behind the state machine and the Raft log, all inserts are linearized, therefore we don't benefit from implementations of the index that are optimized for concurrency.

Production-Ready Implementation. The implementation made in this work is far from production-ready [11]. It suits as a demo and to evaluate the impact and trade-offs of the chosen replication protocol and our mitigation strategies. But with the guidance and reasoning from this chapter and the possible future work in chapter 5, we are very confident that it is possible to move this implementation into a production-ready system, ready to be deployed in a distributed setting, or even on the edge.

3.3.3 Dependability Requirements

The unique features of event stores, compared to other database types are:

- An append-only data structure allows for extremely high throughput rates.
- To be efficient, the data structure requires linearized ordering due to the ordered nature of events.
- Not only writes, but also queries are fast thanks to time-based indexes.
- There is no concept of transactions, every single event write is atomic.
- Only a small fraction of the stored data blocks are written to, while the majority of the blocks are read-only.

In addition to this, ChronicleDB brings with it other special features:

- Aggregates can be derived in-place in the index really fast thanks to small materialized aggregates (SMA).
- Querying speed is extremely fast compared to other event stores and time series databases [10].
- ChronicleDB allows for out-of-order entries and handles them efficiently as long as they do not succeed a certain threshold.
- ChronicleDB allows for fast fail-overs, even in standalone mode.
- Event schemas are fixed and are not allowed to evolve at the moment.

Based on this properties and if we imagine an ideal solution, we can raise the following requirements to a distributed event store:

- **Availability**: The event store is available for writes and queries all the time.
- **Fault-Tolerance**: The event store tolerates crash-faults and is still available and provides non-increasing latency in the face of this faults (up to $n-1$ faults for read availability and up to $n/2-1$ faults for write availability).
- **Reliability**: The operability of the event store is never or at least rarely interrupted.
- **Safety**: Nodes of the distributed event store will never return a false stream of events or false query results.
- **Maintainability**: In case of a failure, it is possible to recover from this failure in a short period of time and without compromising ongoing safety.
- **Integrity**: What has been written can not be changed.
- **Liveness**: Every write to the event store will eventually be persisted.
- **Partition-Tolerance**: Even in the event of network partitioning, the event store is available, and safety is never compromised.
- **Edge-Cloud Readiness**: The event store can be run on all three edge-cloud layers: embedded in another application, standalone on a single node or in a distributed setting, replicated and partitioned on multiple nodes.

Unfortunately, an ideal trade-off free solution can't be achieved. So, what trade-offs are acceptable for us? The trade-offs are described by the consistency model we decide for, which we will look at in the next subsection.

3.3.4 Deciding for a Consistency Model

In this subsection, we elaborate our consistency model decision in detail, based on the previously listed requirements and use cases of a distributed event store, but also on industry best practices as we presented in section 3.2. Subsequently, we will mark the trade-offs to be made. We investigate the concept of consistency from the perspective of real-time and non-real-time event stream consumers and reveal new fundamental properties of consistency in this context.

Throughout this discussion, we will use the notation and definitions from section 2.3.1, plus the following additional definitions:

Definition 3.3.1 (Event Time)**.** The event time describes the time that an event occured, in contrast to the insertion time, that denotes the time an event was inserted into an event stream. We will denote timestamps in event time with t.

Definition 3.3.2 (Read Time)**.** The read time describes the real time of a client that reads an event stream. We denote timestamps in read time with r.

Furthermore, we use the following notation:

- t reflects a single timestamp,
- $e_t \in E$ if the event e_t is included in the event stream E,
- $E = \langle \ldots, e_1 \rangle \xrightarrow{\texttt{insert}(e_2)} \langle \ldots, e_1, e_2 \rangle = E'$ describes a state transition of an event stream.

3.3.4.1 Consistency Perspectives

An event store introduces two different perspectives on consistency:

- The data perspective,
- The client perspective.

While the replication protocol could ensure strong consistency from the data perspective, the system could still look inconsistent from the client pespective due to the allowance of out-of-order events.

This is because strong consistency from the data perspective only cares about

ordering of operations, therefore ordering of `insert` commands, which means it provides a strongly consistent ordering for all replicas by insertion time.

However, in many use cases, clients care about event time. We pay special attention to this case in the further discussion. Figure 3.1 shows the different levels of consistency that can be achieved with strong consistency and allowance for out-of-order events. Figure 3.2 illustrates the two perspectives in the event of out-of-order events arriving.

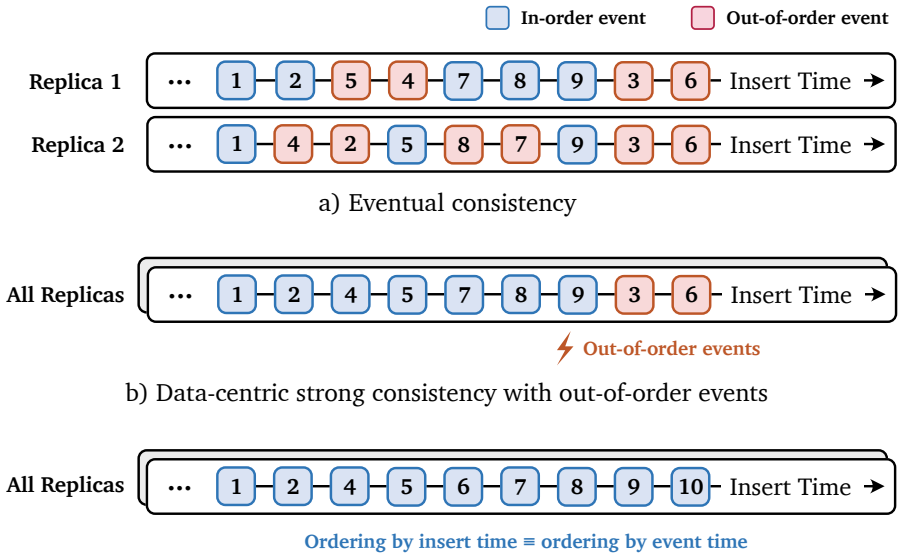


Figure 3.1: Consistency levels for event streams. a) With eventual consistency, no insertion order is guaranteed at all across replicas. b) With strong consistency on the data level, each replica shows the same consistent insertion ordering, but because of out-of-order events, it can still be inconsistent from the client perspective. c) Without out-of-order, strong consistency on the data level provides strong consistency also on the client level, as it guarantees that ordering by insertion time ≡ ordering by event time.

If consistency from the client perspective is required depends on the applications that use the event store. The event store can be used as a *platform* for other applications to build on. In this case, the consistency must be argued from the client perspective. It is not the state of the ChronicleDB state machine that is the critical factor, but the state of the applications that consume one or more event streams from the event store. One event may also be associated with state changes in multiple consuming applications, all with different requirements.

Thus, for event stores, we can describe consistency as the desired trade-off

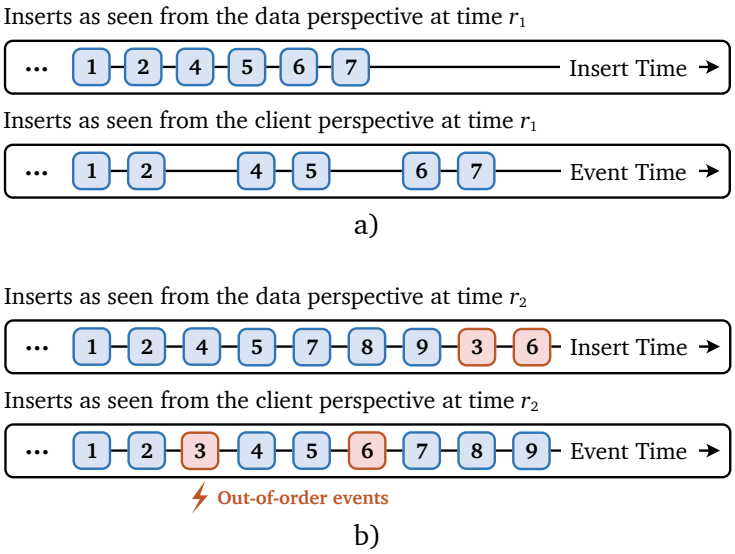


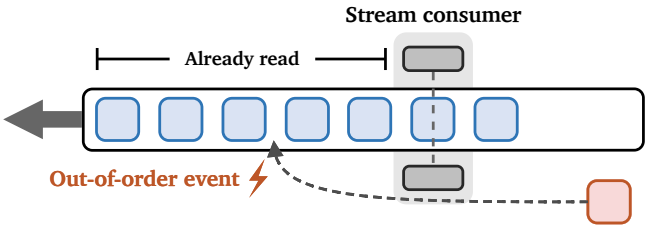
Figure 3.2: Insertion vs. event time. While this insertion ordering in the event of out-of-order events still looks consistent from the data perspective, it looks inconsistent for the client, that only cares about the event time.

between insensitivity to the order of arrival of events and system performance [177].

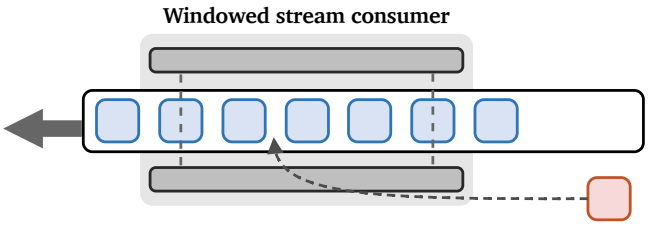
The Problem with Out-Of-Order Events. Out-of-order events naturally occur in real-world streaming systems. This can happen due to network delays, merging of streams with different timestamps and ordering semantics (such as one stream ordered by an "execution end" time while the other is ordered by an "execution start" time—events very often contain multiple timestamps, as is the case in ChronicleDB), or unordered intermediate result streams in event stream processing. ChronicleDB tolerates out-of-order events and allows clients to send events asynchronously, since they can be merged later into their correct position of the stream. In the embedded deployment of ChronicleDB, constant out-of-order rates are considered unlikely, but when it is deployed in a distributed setup, the probability is much higher.

Out-of-order events, if not handled properly, can violate consistency in real-time processing, as late-arriving events can cause non-monotonic derived aggregates to break as the causal chain is interrupted. We will elaborate on this in the course of the following sections.

In the shopping cart example from 3.3.4.2, if the `checkout` event is executed as soon as it arrives, the checkout result will include one item that should have been removed, as it missed the out-of-order `removeItem` event.



a) The consumer will not read this out-of-order event



b) Consumer still able to read this event

Figure 3.3: Real-time stream consumers with time windows. a) As the single event consumer is already past the time of the out-of-order event, it will not read it anymore. b) The out-of-order event occurs when the consumer wants to read the time window, and is therefore still included in the result of the continuous query.

Consistency From the Client's Point of View.

> There's no free lunch in the quest for analytics on both historic data and in real time.
>
> — Jimmy Lin

What are the problems that are tried to be solved with event stores? There are both analytical and operational use cases. In the analytical use cases, real-time and historical data is combined for insight generation. In the operational case, events trigger new events, transactions and operations, such as in event sourcing.

In general, we distinguish between *real-time* and *non-real-time applications* when it comes to stream and event processing[7]. This also introduces two seperate client consistency perspectives: the real-time client perspective and the non-real-time client perspective. A stream that appears consistent to a non-real-time consumer may not appear consistent to a real-time consumer. Commonly applied system architecture approaches provide both real-time and non-real-time processing to users and clients.

[7]Real-time and non-real-time stream processing are also referred to as *online* and *offline* stream processing. We refer the interested reader to the dissertation of Körber on these two types of processing [149].

Two popular architectures that provide this approach are the *Kappa architecture* and the *Lambda architecture*. Both architectures allow for real-time and historical analytics in a single setup. The Lambda and Kappa architectures require that event processing reflects the latest state in both batch and real-time streaming layers (called the *speed layer* in Lambda), hence requiring consistency. A merging layer allows to query the system for both historical and real-time data at the same time. At the heart of these architectures is an append-only data source—such as ChronicleDB, since it allows for both continous and ad-hoc queries[8] [178] [179].

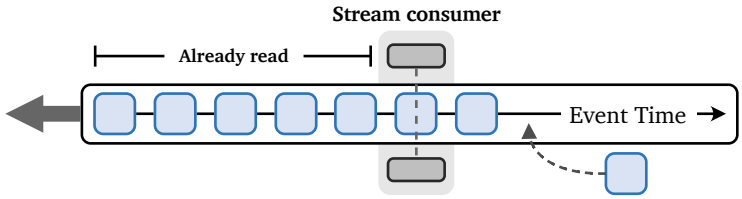
What makes ChronicleDB special here is that it can act both as a stream provider and data lake (or warehouse). It unifies both OLTP and OLAP properties, making it possible to replace multiple lined up systems where data must be stored redundantly (such as Apache Kafka with a Hadoop or Cassandra instance for batch processing) in a Lambda architecture with a single ChronicleDB cluster. Note that currently, ChronicleDB is not able to replace Hadoop when it comes to big data batch processing (MapReduce) that does not fit into memory. ChronicleDB therefore makes an even greater companion in a Kappa architecture, where batch processing is simply done by streaming through historic event data.

Real-Time Applications. In (near) real-time systems, the data is read and processed incrementally, and immediately after it is inserted into the event store. Oftentimes, the result of the real-time analysis of the raw events are aggregates that are served again to other applications that build upon this aggregates, or to end users to query this data, thus relying on the consistency of this data through multiple levels.

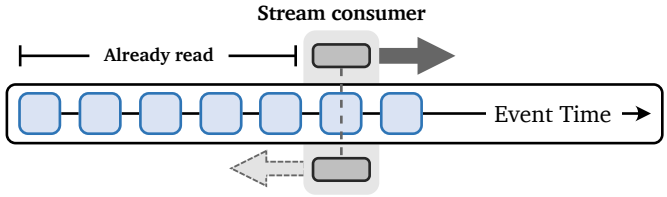
Such systems can be seen in a way that the stream "moves", while the consuming application stays fixed. With the stream moving through the consumer, it sees all recently appended events in near-real-time, at least with some delay. This is shown in figure 3.4 a). Usually, the stream is not scanned event by event, but in time windows, as illustrated in figure 3.5.

Pub/Sub Systems. *Publish/subscribe* (pub/sub) systems are messaging systems that allow for clients to publish events (or messages) into *topics*, while other clients can subscribe to the topics of their choice (often with additional filtering and routing capabilities). Queries to such systems are usually stateless and do not provide powerful computational queries other than filtering. Pub/sub systems like Kafka or

[8] While we reference the blog post of Nathan Marz here which is described as the first mention of the Lambda architecture in 2011 [178], what is in that post should be taken with a grain of salt, as some of the statements made here about "beating the CAP theorem" are simply wrong. Nevertheless, it is still worth a read, as it builds on similar ideas as "Immutability changes everything" by Pet Helland [83].



a) In real-time applications, the stream "moves"



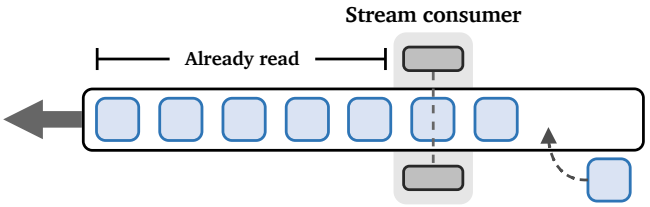
b) In non-real-time applications, the consumer moves

Figure 3.4: Real-time vs non-real-time event stream consumers. a) The event stream "moves", while the consuming application stays fixed. b) The consumer moves along the event stream, probably in both directions, while the event stream stays fixed.

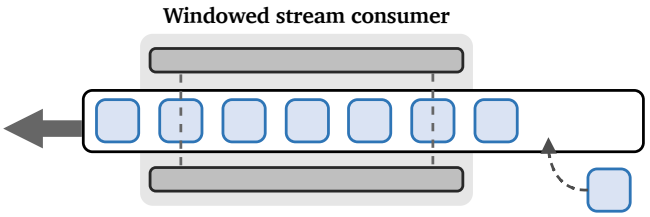
RabbitMQ often allow users to select the consistency level on a per-queue basis.

They often distribute a single message to a magnitude of clients, which is called broadcasting. One example for such messages are mobile push notifications. They rarely need to be sent in order, and arrival guarantees are only eventual in a certain timeframe, but this allows to replicate one message to millions of devices with very low latency. This is an example of optimistic replication with eventual consistency, as the publisher of such a message does not require recipients to acknowledge the receipt—which is nearly impossible as it is highly probably that many of the target mobile devices on the subscribers list are not available. Another example for eventually arriving messages are SMS messages, but in this case, their receipt is acknowledged, and the SMS is sent to a single recipient rather than being broadcasted.

But those systems can also send messages with high safety and consistency guarantees, where both the shipping and receiving of the message is confirmed by a quorum, and the message order is guaranteed at least if a single consumer reads the message. If messages are distributed across multiple consumers (e.g., to distribute load to workers), ordering can not be guarenteed—that is, processing order. In the case of a processing failure, to achieve strong ordering guarantees, the whole queue must be blocked and possibly already started processing stopped and reverted. That



a) Single-event consumer



b) Stream consumer with time window

Figure 3.5: Real-time stream consumers with time windows. a) Consuming one event at a time. b) Consuming multiple elements at the same time with time windows.

is some behavior that is rarely desired.

Due to the lack of more powerful computational queries (which is in fact not their target use case; these systems are designed to route events rather than process them), one often sees these systems combined with event sinks (data lakes), either on the producer or consumer side, or both, which in this case could be ChronicleDB.

CQRS and Event Sourcing. In use cases such as *Command and Query Responsibility Segregation* (CQRS) with event sourcing, commands that are logged in an event store will always be written (and read for consecutive commands) in a strongly consistent manner. However, on the query part, an aggregate is derived (called a *projection*) and shown, that gets stale very fast, due to the high velocity of events written to such an event store. What a user sees on their screen may be already outdated, depending on the domain of that data. This means that users can only be served with eventually consistent snapshots of the current state—even if the UI is reactive (i.e. events are pushed, e.g., via web sockets, to notify the UI about a potential state change), because without locking the stream in the period of loading the most recent state, the state could already have been updated. This happens, for example, in high-volume stock trading, where online brokers try to serve the

current price and order book as recent as possible, but this heavily relies also on the network latency of the user. When placing a market order, it is filed strongly consistent against the most recent price at this time, which the user may haven't yet seen, while some other brokers freeze the last price seen by the user for some seconds to guarantee exactly this price - which is then to be placed on a possibly already updated market.

Event Stream Processing. ChronicleDB is designed to run side-by-side with *event stream processing* (ESP) systems.

Traditional ESP applications handle a single stream of events that arrive in the correct event order. An example would be algorithmic trading, where a simple ESP application could analyze a stream of price data and decide whether to buy or sell a stock, such as in figure 3.7. ESP applications typically do not involve event causality or event hierarchies.

Typical event stream processing queries look relatively similar to SQL. In the *Continuous Query Language* (CQL), a query looks similar like this:

```
SELECT AVG (O.total)
FROM Orders O [RANGE 10 Minute, SLIDE 10 Minute]
WHERE O.product = productId
```

This query returns a moving average of the order volume for a certain product over a tumbling time window of 10 minutes.

In ESP, we differentiate between in-order processing and out-of-order processing.

IOP (In-Order Processing) vs OOP (Out-Of-Order-Processing). *In-order processing* (IOP) systems require strong consistent ordering of the event streams that are continously read. In case the events from the event store are to be fed into such systems, the event store must ensure strong consistency.

This approach fits all use cases where strong consistency is required, such as use cases from the financial domain or other real-time decision making cases with high business criticality.

On the other hand, *out-of-order processing* (OOP) systems do tolerate such events arriving with a delay at the processing layer. This can speed up the processing of such systems tremendeously, especially in distributed event processing, in a similar fashion that eventual consistency speeds up the replication in a distributed data storage system and reduces overall latency. The windowing techniques of such systems allow for a timeframe where out-of-order events can appear without violating the

results of the event processing. They can be used for all real-time applications where the use case allows for a degree of inconsistency. Once the use case allows for OOP, this approach can "significantly outperform IOP in a number of aspects, including memory, throughput and latency" [180].

An out-of-order processing system continues to process a stream until a specified condition applies. While the ordering guarantees of IOP allow the progress of stream processing to be acknowledged after each processed in-order event, in OOP, progress is confirmed only when such a condition occurs. These conditions can be expressed using *punctuation*.

Punctuation works similar to punctuation in written language and marks the end of a substream (analogous to the full stop marking the end of a sentence). This allows us to view an infinite, unbounded event stream as a mixture of finite streams [181]. Inside of such a sequence, order does not matter. No further event violate previously received punctuation. OOO is allowed to happen in such a "sentence", but not across sentences. Punctuation provides a guarantee to a stream that all events arriving after the punctuation will have an event timestamp greater than that of the punctuation.

With punctuations, only the punctuations must be ordered, and events between the punctuations are allowed to be unordered, but must be the same set of events (no out-of-order events are allowed to come after their punctation). To identify the correct set of a sequence, the punctuation message `punctuate({e_1, e_2, \dots})` contains the ids of those events to expect in the sequence. This is similar to the "weak consistency" approach, where only expliticitly synchronized parts must appear in order, and everything in between can be unordered. This means that for all events e with $e.t \geq p_1.t \wedge e.t < p_2.t$, $p_1 \prec e \prec p_2$, while any order between events is valid, where p_1 and p_2 are the punctuations around e.

In ChronicleDB, a similar approach could be adopted to mitigate the out-of-order problem. With this approach, events would not be required to be inserted in event order, while the punctuation allows for strong ordering guarantees in the aftermath. It would also require event producers to support punctuation. There is literature covering the case that event producers can not produce punctuations themselves, which requires the stream processor or event store to deduce these themselves (called a *heartbeat* then) [182]. Punctuation processing could either be implemented right in the replication protocol by introducing it in the log and weaking the consistency guarentees between punctuation, or in the event streams themselves. We haven't figured out how this could be implemented, though (what if punctuations arive themselves out-of-order?).

Not all use cases can be served with OOP, as implementing and applying punctuation is far from trivial and can not be used in every situation. Nethertheless, this approach comes in handy for all real-time applications where no strong consistency is needed, such as calculating moving averages and similar aggregates.

Non-Real-Time Applications. In non-realtime applications, the consumer moves along the event stream, while the event stream stays fixed. The consumer can move in both directions, scanning the stream. This is shown in figure 3.4 b).

In such applications, out-of-order events are not a big threat, as in general, they arrived a long time ago in the event store when a query is run on historic data. Even more, the eventual consistency model is oftentimes sufficient since it is only necessary that the write operations converge at some point in time in the event store.

Examples for non-real-time applications are *Business Intelligence* (BI), data science, process mining or machine learning. We have drawn more examples in figure 3.44. Also, using an event store as a data lake to store derived events after processing does not require strong consistency.

Historic queries can be used to detect faulty results from real-time queries due to delayed events, as the results of historic queries have a higher chance to be consistent. To repair such a faulty result, it requires the developer to write compensation logic in case the real-time layer already caused executions based on incomplete or stale data, similar to the *Saga pattern*. But, this is only limited to actually compensable operations. For example, how should a business compensate financial transactions in the real world that are made based on wrong assumptions due to incorrect data?

Complex Event Processing. *Complex event processing* is a method to derive new conclusions from event streams in real-time. It applies pattern matching to derive new events, called *complex events*, from a sequence of raw events [1]. From a domain-driven perspective, these complex events are *derived aggregates*.

Compared to ESP, complex event processing systems patterns are generally detected across multiple streams of different domains. We illustrate this in figure 3.6. Complex event processing systems can detect patterns across event streams, including event occurrence and non-occurrence, and queries can set complex constraints on timing. An example for a pattern that matches against the non-occurrence of events is one that detects customers that did not respond to an email within a certain time frame.

To run patterns that rely on the non-occurrence of events (thus, deriving non-monotonic aggregates), a strong consistent ordering during the pattern matching

windows is required. In general, CEP systems do not provide strong consistency guarantees, which puts the demand for ordering on the event producer (or, in this case, the intermediary store). We have illustrated what can happen when events appear delayed in figure 3.7.

Eventual consistency for CEP is only allowed for monotonic problems, because in this case new information does not change the already derived facts. Additional facts can only arise from discovering additional derived events: the output (computed events) grows monotonically with the input (factual events).

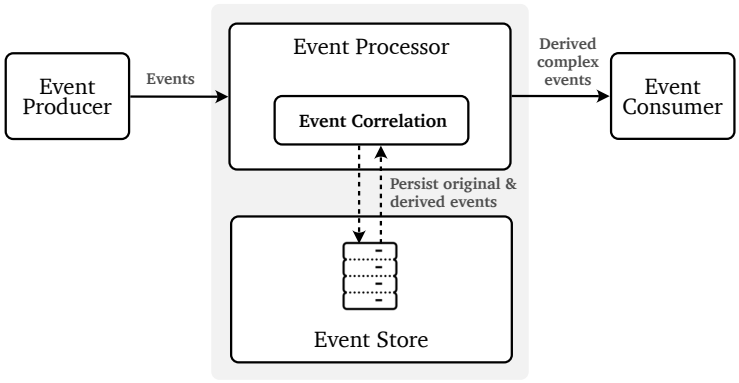


Figure 3.6: In complex event processing, events from multiple streams are fuzzily correlated to derive monotonic aggregates

Complex event processing is also a key component of event-driven systems. Instead of dealing with raw events, microservices consume complex events (such like a signal) derived from these raw events, and they must do this in a reliably, safely manner.

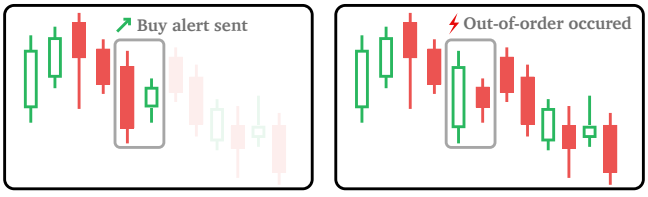


Figure 3.7: Algorithmic trading as an example for strong consistency requirements. a) A candlestick pattern was detected in the trading events in a order book, causing a buy alert to be sent (and executed). b) At a later time, the past candlesticks are displayed differently due to some out-of-order events, so that the buy alert is now wrong - but it is already too late. This reduces trust in this hypothetical trading platform and its reputation.

Other Use Cases. The event store can be simply used as a replication engine itself (similar to ZooKeeper) by running it next to a distributed system to store and replicate the commands of this system. It then turns out to be a replicated log with some extensions (timestamps denoting event-time, aggregates and powerful indexing). It could also be used to create event-sourcing system or to create a pub-sub architecture. For the consuming application(s), the consistency of the system is then also dependent on the allowance for out-of-order events. If the use cases of the consumers allow with eventual consistency, then a eventual consistent implementation that also allows out-of-order events is sufficient. Else, the system must be linearizable, providing strong consistency, which also forbids any out-of-order events.

Conclusion. So far, it appears that some applications require a strict notion of correctness that is robust in terms of the event order, while others place more emphasis on high throughput and low latency. The perspective of the application is more important than the perspective of the event store when it comes to a consistency decision. We need to remember that a recently inserted event may be associated with state changes in one or more consuming applications, as in figure 3.8.

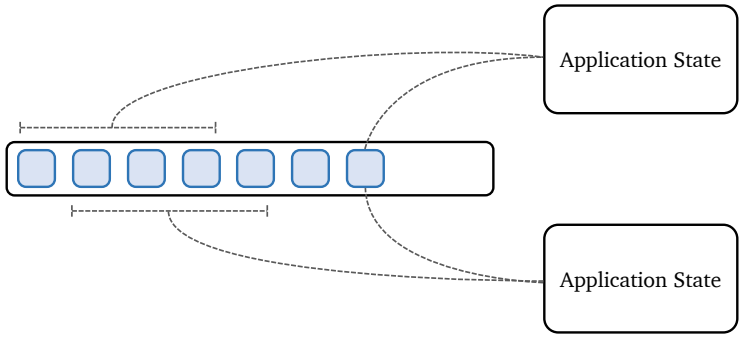


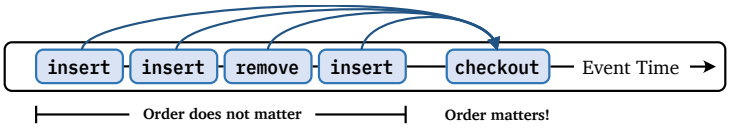
Figure 3.8: Multiple applications that derive their state from subsets of an event stream

3.3.4.2 Causal Consistency

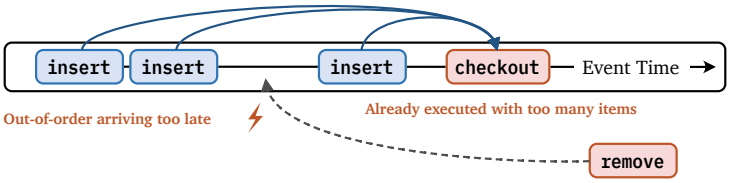
In an event store, causal consistency must be evaluated from a client perspective, since the only write operation is the `insert`, and there are no mutations of existing events. Events can have intra-stream as well as inter-stream causal relationships, and these relationships must not be explicitly modeled between stream [183].

One example for causal relations between events comes from the event sourcing domain: remember the shopping cart example from section 2.1.4.3. Even if

`insertItem` and `removeItem` commands can be designed to be monotonic, thus non-causal from a data perspective since their order does not matter for each other, a `checkout` command is causally related to all these previous commands for this particular shopping cart session, thus it must be ordered after this commands. In event-sourcing systems, such commands should never arrive out-of-order if executed in realtime, because it could render the derived aggregate wrong that defines the items included in the checkout. We illustrated this in figure 3.9.



a) Causally related operations in event sourcing



b) Causally related events not allowed to be inserted out-of-order

Figure 3.9: Causally dependent operations in event sourcing, illustrated with a shopping cart example. a) While the order between some domain events does not matter, the order between them and another command is important, implying a causal dependency. b) If causally related events arive eventually, such as with out-of-order events, an already executed command may have produced wrong results.

The causal relation of events can depend on attributes of the events and their domain (note that we are not discussing causal consistency from the perspective of write-read command chains as in the actual definition from section 2.1.4, but from real-world causality). For example, if a single stream contains events from multiple sensors, and these sensors do not influence each other, a sensor ID can be used to segment the stream. The happens-before relation applies to all events of such a segment, thus the events in the segment are causally related. This is illustrated in figure 3.10. In another example, these sensors may be measuring temperatures in different regions, but the client is interested in finding correlation in temperature changes between these regions, but also with temporal causality: does the change in temperature at one location affects the other?

But there can also be a causal relationship across streams with different schemas, depending on the domain, as illustrated in figure 3.11. For example, consider

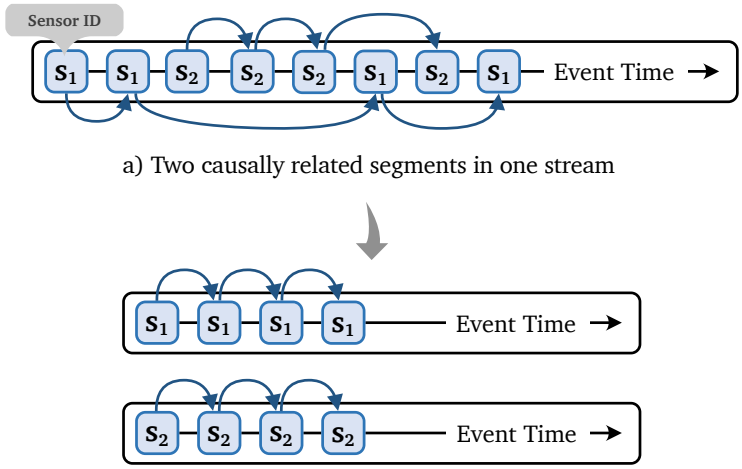
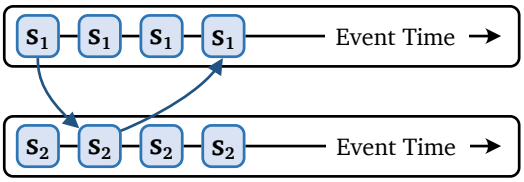


Figure 3.10: Two causally consistent segments of event streams. a) Both segments reside in a single event stream. b) The segments are split up into their individual streams, that are now strongly consistent.

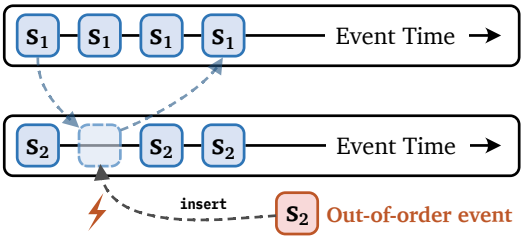
an event stream that contains temperature measurements while another contains humidity measurements. The events between the streams are assumed to be causally related: A rise in temperature causes a (perhaps delayed) rise in humidity in the same region. Finding such patterns is a typical use case of complex event processing (CEP). Note that causal consistency is not sufficient to detect patterns that include the non-occurrence of events: causal consistency only covers the observation of existing events.

Since this is up to the user, we recommend to design streams in such a way that the causal relationship is kept coherently on a per-stream base. That is, if a stream can be segmented in a way that the segments are no longer causally related, those segments should be streams of their own, and if events appear to be causally related across streams, it may be appropriate to merge the streams, provided their schemas are compatible. This also increases the performance of the system, as `insert` operations of unrelated events do not need to be coordinated. However, some other properties must be taken into account, such as data locality (keep seperate streams close to where they are created), edge-cloud design (merge streams of two sensors afterwards on the edge or cloud if they are causally related) and sharding (find a balance between causality and scalability; i.e. if the throughput limit of one stream has already been reached, it makes sense to partition it even if events in these shards

are causally related).



a) Inter-stream causality



a) Intermediate inter-stream causality violated due to out-of-order events

Figure 3.11: Two event streams, i.e. for different domains. a) Inter-stream causality between events of the two streams. b) Due to out-of-order events, intermediate states of the streams violate causal consistency.

Out-Of-Order Events and Causality. Out-of-order events cause violations of intermediate states of streams, as the causality chain can be broken. This is illustrated in figure 3.11 b). Depending on the consuming applications, this can be a serious problem. Figure 3.12 shows a generalized picture of how non-monotonic aggregates, which are apparently causally dependent on the source events, can be violated by out-of-order events. Such faults introduced by delayed events would require read-repair code, such as applying the SAGA pattern, to be mitigated in the aftermath, which is prone to more errors to be introduced, and even not always possible.

The following sections describe the out-of-order problem in more detail and explain how it can be mitigated.

Conclusion. Many use cases require a distributed event store to be at least causally consistent, but also with respect to the user perspective. Unfortunately, the causal relation between real-world items is unknown to a replication protocol; a protocol only knows about writes and reads and the causal dependency between these

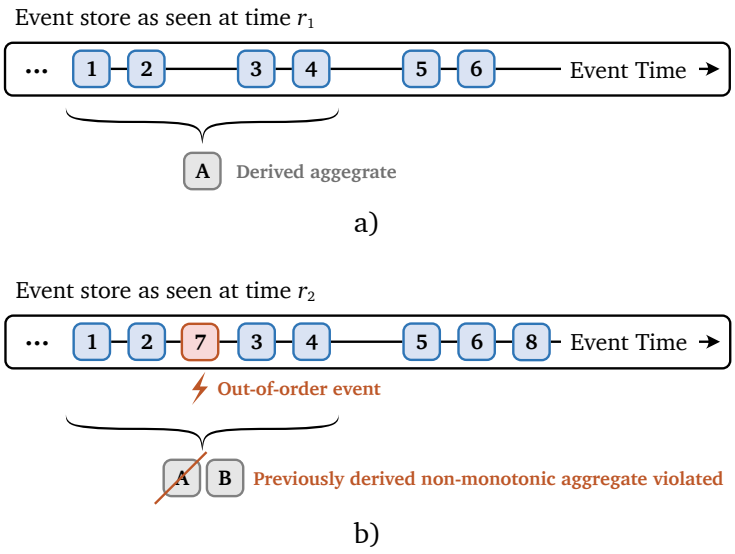


Figure 3.12: For non-monotonic derived events or aggregates, out-of-order events must be prohibited, as they could render the derived events wrong: what was already observed must not change. a) A non-monotonic derived aggregate is created based on previous facts (events). b) When observing the event stream later again, an out-of-order event occured which invalidated the previous derived aggregate and created a new aggregate.

commands. This must be mitigated by the user by proper design of the event stream partitions.

3.3.4.3 Coordination-Free Replication

One of the first questions to ask when thinking about a consistency model is, if we can achieve coordination-free replication—which means, eventual consistency is sufficient. We ask ourselves if event stores are monotonically growing, which is the requirement for coordination-free replication as described in the CALM theorem (see subsection 2.1.4.3).

So far we have seen that for many use cases, causal consistency is needed, which means that coordination-free replication is not possible in this cases. In this subsection, we elaborate this in more detail.

Currently, there is only one write operation in the event store: the `insert` operation. We consider ChronicleDB to be append-only, therefore ignoring the case for deletes. In fact, most cases of deletes are better represented as creating new events that invalidate or reverse the state change of the the initial event (if deletes are actually required, such as to adhere to the GDPR, it gets more complicated).

From the point of view of a single event, the `insert` operations is associative, commutative and idempotent, all the properties that allow for coordination-free replication with eventual consistency:

- Inserting a single event at a given timestamp will insert it at this timestamp no matter if the insert was executed before or after some other insert.
- Using mechanisms like a session identifier, repeated inserts of the same event at the same timestamp could be ignored, rendering the insert operation as idempotent.

But from the point of view of the overall state, which is a ordered streams of events, the insert is not associative and commutative: Running inserts in different orders will result in different intermediate states of the stream, even if the final state is the same. Imagine inserting two events e_1 and e_2 in any order, this could result in two intermediate states $\langle \ldots, e_1 \rangle$ and $\langle \ldots, e_2 \rangle$, while the final state is $\langle \ldots, e_1, e_2 \rangle$. $\langle \ldots, \texttt{insert}(e_1), \texttt{insert}(e_2) \rangle$ leads to a sequence of states $\langle \ldots, e_1 \rangle \rightarrow \langle \ldots, e_1, e_2 \rangle$, while $\langle \ldots, \texttt{insert}(e_2), \texttt{insert}(e_1) \rangle$ leads to an inconsistent sequence of states $\langle \ldots, e_2 \rangle \rightarrow \langle \ldots, e_1, e_2 \rangle$. Without any other hints, applications may not be able to distinguish between an intermediate and final state, nor should they be prompted to do so. We could lower the requirements and help applications distinguishing between an intermediate (unordered and inconsistent) and final state with punctuation as shown before in this chapter. We then let query operators order punctuated sequences before returning them, as shown in figure 3.20). Derived aggregates that are only allowed to be created for punctuated sequences are safe then. But introducing punctuations is far from trivial. For instance, a punctuated sequence would look like the following: $\langle \ldots, \texttt{insert}(e_2), \texttt{insert}(e_1), \texttt{punctuate}(e_1, e_2) \rangle$ leads to a sequence of states $\langle \ldots, e_2 \rangle \rightarrow \langle \ldots, e_1, e_2 \rangle \rightarrow \langle \ldots, e_1, e_2, \cdot \rangle$, where only the last state is considered to be a final state.

Conclusion. Depending on the use case, an event store is not monotonically growing. The derived aggregate resulting from the event streams may or may not be a monotonic aggregate, depending on the application using the aggregate. With non-monotonic aggregates, the order of the events, and also the order of the appearance of the events, matter.

The consistency requirements to the event store are heavily dependent on the actual use cases of the consuming applications.

At least causal consistency is required by a subset of real-time applications that rely on non-monotonic aggregates. Punctuations could be a way to introduce causal

consistency and to reduce coordination inside of punctuated sequences. Non-real-time applications will probably not experience missing events due to out-of-order or eventual consistency, so eventual consistency is oftentimes sufficient.

3.3.4.4 Time-Bound Partial Consistency

When out-of-order events can not be avoided, or the maximum throughput should be increased (and latency reduced) by introducing lower consistency constraints, it is worth to think about *time-bound partial consistency*. This means that for different sequences of a event stream, different consistency promises could be made. This allows to bypass the latency-consistency trade-off of the PACELC theorem, at least practically. Figure 3.13 shows that we for older time windows, consistency increases—in general in an exponential nature—and we want to take advantage of this property.

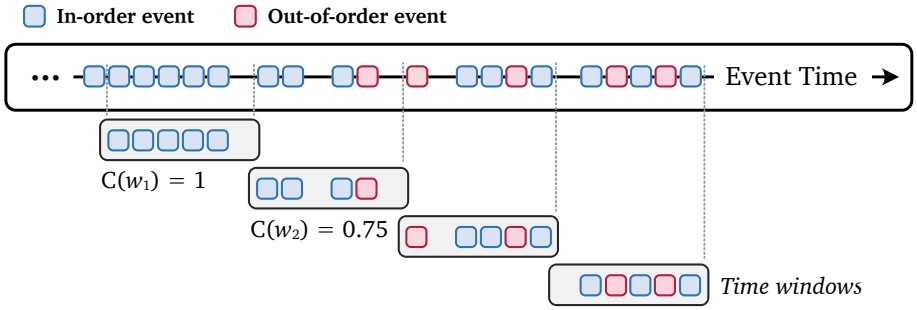


Figure 3.13: Illustrative example for increasing consistency for past time windows

Convergence and Eventual Consistency. As we have shown, strong consistency can be relevant for real-time applications, contrary to the low-latency requirements for real-time processing. Non-real-time applications will probably not experience missing events due to out-of-order or eventual consistency, as the probability that time windows of the event stream provide a consistent view—i.e., that the have converged—increases when we move along the stream in descending time order, and it also increases with a decreasing window size.

By investigating the convergence behavior of a system, we can make better assumptions on its practical consistency. Therefore, we will describe consistency and convergence in the following definitions:

Definition 3.3.3 (Ground Truth). $E^{G}_{[t_i,t_j)}$ describes the total series of events that

happened during the time interval $[t_i, t_j)$ and that are to be persisted in the event store. We call it the ground truth of events during this interval.

We expand our definition of time windows:

Definition 3.3.4 (Time Window at Read Time r)**.** $E^r_{[t_i,t_j)}$ refers to the time window $E_{[t_i,t_j)}$, observed at read time r. The read time is the real-time where the time window is observed, i.e., by a consumer.

Definition 3.3.5 (Consistency)**.** With $C(E^r_{[t_i,t_j)})$, $C : \mathscr{E} \rightarrow [0,1] \in \mathbb{R}$ we denote the consistency of the given time window at real-time r. It denotes the ratio of existing and known events that are in the time window of the stream at this given point in time:

$$C(E^r_{[t_i,t_j)}) = \frac{|E^r_{[t_i,t_j)}|}{|E^G_{[t_i,t_j)}|}$$

Eventual consistency guarantees liveness: data will eventually converge in the absence of updates. We say that a stream converges if

$$\lim_{r\rightarrow\infty} E^r = E^G \tag{3.1}$$

where r refers to read time. The same applies then to each possible time window:

$$\lim_{r\rightarrow\infty} E^r_{[t_i,t_j)} = E^G_{[t_i,t_j)} \qquad \forall t_i, t_j, t_i \leq t_j \tag{3.2}$$

Based on 3.1, it is

$$\lim_{r\rightarrow\infty} C(E^r_{[t_i,t_j)}) = 1 \qquad \forall t_i, t_j, t_i \leq t_j$$

In the absence of updates.

We claim that this is also the case for a fixed time window during consistent updates up to a certain maximum throughput.

We need to describe convergence first:

Definition 3.3.6 (Convergence Rate)**.** The convergence rate `Conv` of an event stream time window $E_{[t_i,t_j)}$ in the real-time interval $[r_n, r_m]$ describes the rate of events of the ground truth E^G inserted into the stream during the interval, compared to new events that arrived in the ground truth.

The convergence rate is defined as

$$\texttt{Conv}(E_{[t_i,t_j)}, [r_n, r_m]) = C(E^{r_m}_{[t_i,t_j)}) - C(E^{r_n}_{[t_i,t_j)})$$

Hence, it can be generalized to the convergence rate at read time r

$$\texttt{Conv}(E_{[t_i,t_j)}, r) = C'(E^{r}_{[t_i,t_j)})$$

A local convergence rate > 0 means the stream is about to converge, while a local rate of 0 describes a stream that does not converge at the moment (either it already has converged or the gap size between inserted and occurred events stays consistent), while a local rate < 0 describes local entropy (there are more events occuring than the system is able to handle).

We observe that a fixed time window becomes eventually consistent if and only if the average convergence rate ≥ 0.

$$\lim_{r\to\infty} E^{r}_{[t_i,t_j)} = E^{G}_{[t_i,t_j)} \longleftrightarrow \texttt{Conv}(E_{[t_i,t_j)}, [r_0, r]) \geq 0$$

Convergence and Scaling. In real-time systems, writes and reads occur continously. Until writes stop, an event stream will never converge, as there are always new events occuring that are not yet written to the store.

Looking at a fixed time window, it will only never convergence if and only if delayed (out-of-order) events are never written to them. If this happens, there was either a fault that prevented the event from appearing, a extremely long network delay, or, the most problematic case, the overall throughput is too high, lowering the convergence tremendously. Regular convergence behavior of a fixed time window is illustrated in figure 3.14. We will see in the next paragraphs, that we can describe a probabilistic guarantee for time windows to converge.

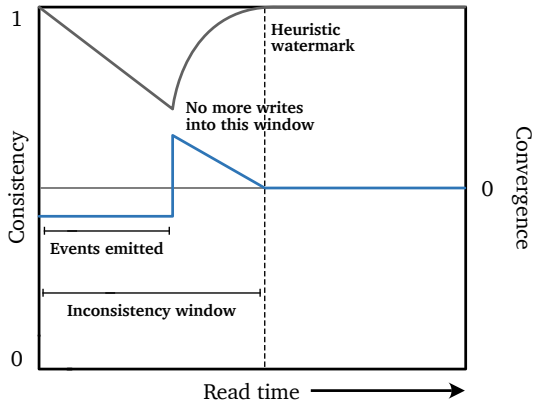


Figure 3.14: Looking at the time window at the head of the stream, we see that it converges after a period of inconsistency when events, including out-of-order events, arrive at the window

Continously looking at the head of the stream (the most current time window), under continous writes it will never convergence. If the throughput is acceptable, we will observe a steady convergence rate of 0 (if we write in batches, we may see some stairs-shaped curve). We use this observation in the following paragraphs to describe a way to improve write latency and give better consistency guarantees for reads.

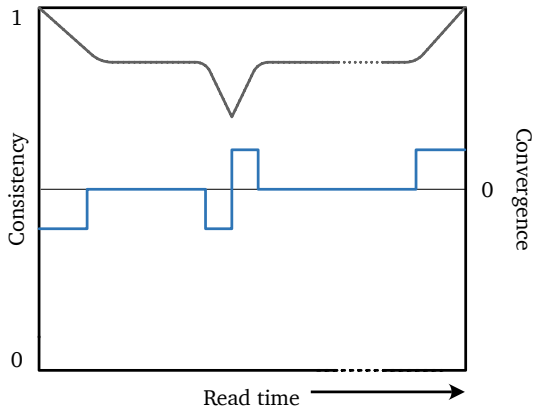


Figure 3.15: Simplified illustration of the convergence behavior of the head of an event stream. Continously looking at a time window at the head of the stream (i.e. moving it while we move in real-time), we see that the head of the stream never converges until writes to the stream stop. We can identify throughput spikes. If the average convergence rate is 0, the throughput at this stream and node is healthy.

A continously negative convergence rate is a cause for concern and signaling that the current throughput may be too high for the system to handle, causing entropy between nodes and clients and finally slowing down the system. This could serve as an ideal signal for elastic auto-scaling: When entropy is detected, new nodes can be provisioned and added to the cluster to handle and balance the additional load. But since only the producing client knows the ground truth of events, it must be responsible to detect convergence by comparing what is written and incoming write acknowledgements. This won't work in an asynchronous fire-and-forget mode, though.

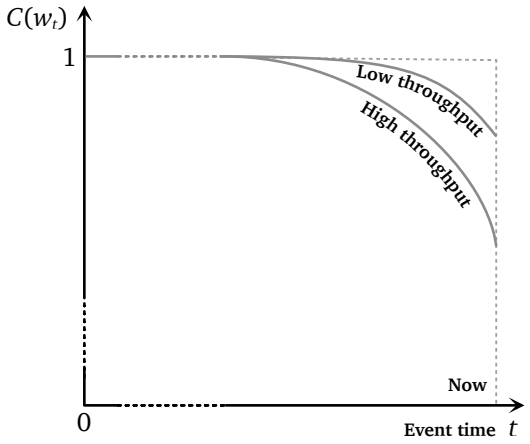


Figure 3.16: Two illustrative consistency functions over time windows for a high and a low throughput scenario. Because out-of-order events are distributed in an exponential manner across a stream as the chance for a time window to converge increases over time, the consistency increases when we travel back in the stream. The rate of out-of-order events increases with throughput—and also with the level of concurrency.

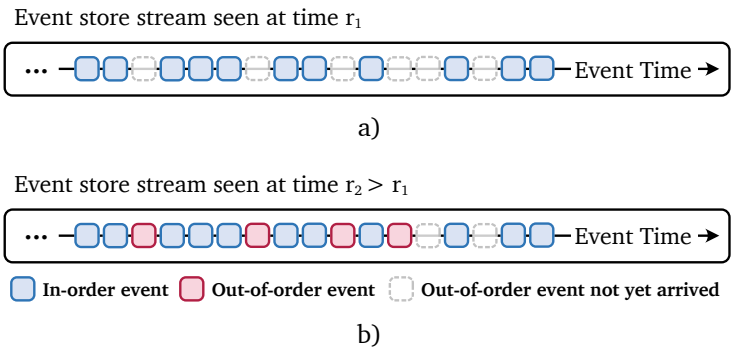


Figure 3.17: Event stream with out-of-order events arriving later in real time, thus only providing eventual consistency to the client. In general, out-of-order events are distributed in an exponential manner, i.e., out-of-order events happen closely to their source. a) The stream is missing some events from the ground truth that are not yet sent to and committed by the store. With decreasing event time, the stream becomes more consistent. b) Eventually at a later point in real time, a past portion of the stream converged, while for a recent time period out-of-order events may still arrive.

The Inconsistency Window. Time-bound partial consistency can be achieved by ensuring that out-of-order events (or all delayed events in the case of eventual consistency) are written only in a certain time window, called the *inconsistency window*. We sketched to inconsistency window in figure 3.14. Depending on the use cases and business criticality, this can be of high interest, as the cost of replication (such as latency) can be massively reduced.

Let the consuming client decide on the level of allowed inconsistency by specifiying the latest timestamp $t_{\text{now}-n}$ to that events are returned. This timestamp is referred to as the *heuristic watermark* [179]. All events after this timestamp (thus, more recent entries) are not returned to the consumer, as this time window is allowed to be inconsistent.

With out-of-order events, and especially with eventual consistency, there is no guarantee for consistency. This means that the timestamp is chosen in a way that $P(C(E_{[t_{\text{now}-n},t_{\text{now}}]})) \leq 1$, but $P(C(E_{[t_i,t_j]})) \approx 1 \; \forall t_i, t_j, t_i < t_j < t_{\text{now}-n}$. We denote $E_{[t_{\text{now}-n},t_{\text{now}}]} = E_{|n|}$ as the *inconsistency window*.

The inconsistency window is either not allowed to be read by consumers or the consumers must explicitly require it, knowing that they may receive inconsistent results.

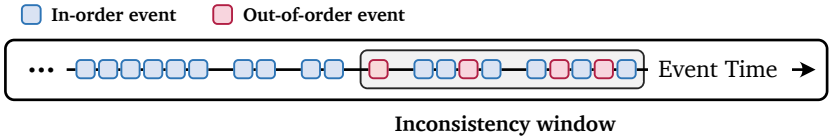


Figure 3.18: Illustration of the inconsistency window. The events in this window are not immediately visible to consuming clients because out-of-order events are likely to appear here.

Definition 3.3.7 (Out-Of-Order Probability)**.** The out-of-order probability $P_{\text{OOO}}(E_{[t_i,t_j]})$ describes the chance for one out-of-order event to appear in the stream in the time interval $[t_i, t_j]$. We illustrate this in figure 3.19.

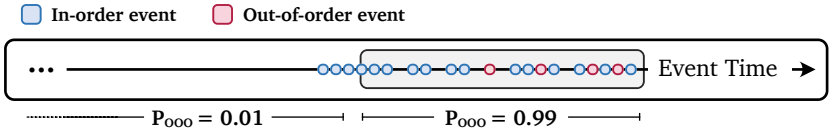


Figure 3.19: Illustration of the out-of-order probability for the inconsistency window and the remainder of the stream.

Definition 3.3.8 (Inconsistency Probability)**.** The inconsistency probability of a time window describes the probability of that window to be inconsistent, thus $P_{\text{IC}}(E_{[t_i,t_j]}) = 1 - P(C(E_{[t_i,t_j]}))$.

In an event store with strong consistency, this is equal to the out-of-order probability.

The out-of-order and inconsistency probabilities can be estimated by continously observing an event stream and are served on a per-stream base. In the non-faulty case, the maximum size of the inconsistency window can also be estimated based on communication delays, system load, and the number of servers involved in replication [184]. ChronicleDB supports calculation of the *maximum delay* for events on the fly. It describes the maximum amount of events that are present between an out-of-order event and the place that it was supposed to be inserted at according to application time [10]. This allows to derive the inconsistency window with built-in methods.

With this, rather then deciding for the heuristic watermark, a real-time application could agree on a tolerated inconsistency probability. This means that they would define the minimum required consistency probability p_C for the remainder of the stream (the sequence of the stream that is allowed to be read) by configuration.

This is equal to the the minimum inconsistency probability that the inconsistency window must cover, making it simple to argue about the length of the inconsistency window (and to derive the heuristic watermark):

We continously observe and register the delays of out-of-order events in the stream and set the window size in a way that it covers the p_C percentile of these delays, including an additional tolerance area (such as 20% of this timeframe). We choose the percentile so we can exclude outliers, as shown in figure 3.21. We don't recommend to choose a value of 100%, as if any outlier occurs (due to a node fault or other extreme delays), it would render the inconsistency window too long, thus denying any near-real-time properties.

This estimate works well enough when the out-of-order events are exponentially rather than uniformly distributed, i.e., out-of-order events happen rather closely to their source. This is generally the case. We illustrated this in the figures 3.21 and 3.17. We advice to continously monitor the arrival of out-of-order events (or, in eventually consistent streams, the time for events to arrive in general) to be able to react on expectional patterns that might render the time windows before the window inconsistent. The inconsistency window should be calculated on a per-query base, as shown in figure 3.20.

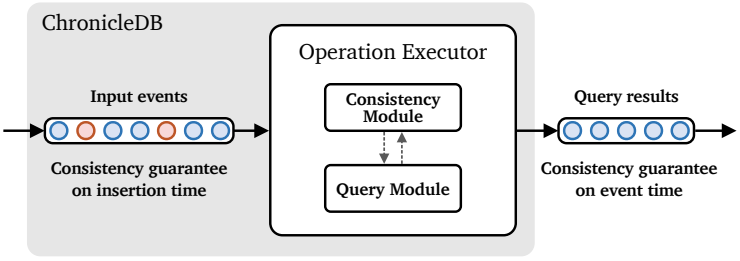


Figure 3.20: Potential query consistency module for ChronicleDB. An intermediate consistency module could provide practical ordering guarantees for queries by adding additional delay on a per-query base that incorporates the maximum delay for out-of-order entries in the inconsistency window. Practically, the module holds the input stream in a buffer until an output can be generated that adheres to the desired consistency level. Applications must therefore make trade-offs between delay and consistency for each query.

When deciding for the consistency probability, clients need to find a balance that allows for a low chance of inconsistent entries to be read, while available data should be as recent as possible (near real-time).

Guaranteeing the Inconsistency Window. In theory, the whole stream is still eventually consistent from the clients view, but in practice, inconsistent reads outside

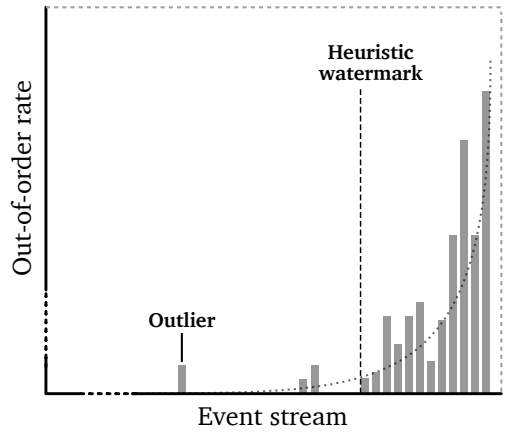


Figure 3.21: Distribution of out-of-order events in an event stream, relative to the current timestamp

the inconsistency window should be very rare. The consistency probability could be turned into a guarantee by covering it in SLAs, for example. Or, we could even harden this property by requiring the following guarantee: All events must be eventually written during the inconsistency window. This would mean that we would reject any out-of-order entry that reaches the event store with a delay larger then the inconsistency window, sending a `nack` to the event producer and making the producer responsible to compensate for the missing event.

Conclusion. By investigating the convergence behavior of a stream, we can argue about different levels of consistency throughout one stream from the clients perspective. By continously observing convergence and adjusting the inconsistency window on a per-stream basis, we can improve the latency and consistency while providing near-realtime reads to consumers. From a theoretical point of view, this still does not provide a strong consistency guarantee, but the probabilistic guarantees may be sufficient for several real-time use cases where write latency is critical.

3.3.4.5 Buffered Inserts

One solution to cope with out-of-order events is to buffer `insert` operations temporarily before writing them to the stream. The buffer is flushed and all events written in a batch once its content reaches a certain size or after a certain timeout after the last write. On flush, events in the buffer are ordered by event time, instead of insertion time, ensuring consistency for the batch write. This also decreases

latency introduced by the replication protocol dramatically, since the network round-trips are reduced, which we will elaborate later in this chapter. We found similar approaches in the literature [182].

The the buffer size and timeout can be chosen in a way that it covers the max delay of out-of-order events—respectively the inconsistency window. This has a similar effect to implementing the inconsistency window in the query processor, with the difference that events in the buffer are only hold in-memory on a single machine instead of being written directly to the event stream, thus causing the risk of data loss of the events in this buffer. While in our approach, the user must define the buffer size, to guarantee consistency, the buffer size should be adjusted according to *progress requirements*.

The effect of a write buffer on the overall consistency is shown in figure 3.22.

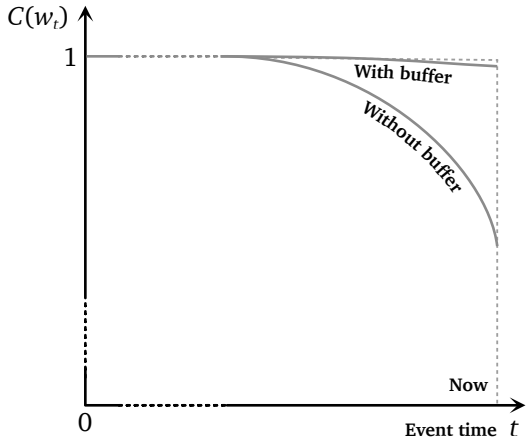


Figure 3.22: Using a buffer to order events in advance and write them in batches increases the consistency dramatically in the face of out-of-order events and concurrency.

While a buffer increases the real-time consistency from the client perspective (the chance is higher that events arrive at consumers ordered by event time), it reduces the data consistency from linearizable to sequential, since after a write, an event can not be immediately read. But, when it is read in real-time, it is ordered (with a high chance). Acknowledging each single write to the stream to the producer is challenging: this would require callbacks that must be managed somewhere, e.g. in the replication log. We currently only acknowledge a successful write to the buffer, but this does not guarantee a successful write to the quorum. Only the acknowledge of a flush of the buffer and the execution of the batch of events lets the client know of a successful write to the store, but this adds a lot of latency to

this acknowledgements. In case of unavailability or faults, the client will not get an immediate insight into this.

3.3.4.6 Consistency Across Partitions

Keeping writes consistent across partitions, and especially across streams of different schemas, is challenging, especially when each partition is covered by its own instance of the replication protocol. This requires another layer of coordination across partitions, which increases the overall latency. It is also only relevant for streams with inter-stream causality (see subsection 3.3.4.2).

Instead of linearizing writes across partitions (which anyway only ensures ordering in insertion time), it is better to ensure inter-stream ordering on query time. If there is causality across streams, this becomes only relevant on queries on multiple of such streams, and should be handled by the distributed query processor.

Note that even popular event stores and similar systems often do not support strong consistency guarantees across partitions, such as in Azure Event Hubs [173].

3.3.4.7 Conclusion

What have we decided for? The previous discussion has shown that the consistency model applicable heavily depends on the use case of the event store: event sourcing requires at least causal consistency, ESP relies commonly on strongly consistent data and in CEP, pattern matching also works on monotonicly derived aggregates on eventual consistent data. Non-realtime applications are usually satisfied with eventual consistency. And there are also applications that are satisfied with a strong ordering by insertion time. This shows that the event store is not the application, but a platform, inheriting its consistency requirements from the consumers. We must keep in mind that an event in a stream may be associated with state changes in multiple consuming applications with different requirements. Similarly, the literature on event processing emphasizes that the degree of consistency is highly dependent on the use case and end-user requirements [177].

Let the User Decide. In a one-size-fits-all event store, the user should be able to decide for the consistency model. However, the complexity of such an implementation is beyond the scope of this thesis. We focus on one particular consistency model in the implementation of our demo application and highlight its trade-offs. wAt the same time, we refer to the insights gained in this section to develop such an application with multiple consistency layers that allows the user of the event store

to decide on consistency, e.g., on a per-stream basis.

We refer to Gotsman et al. [79] to help deciding for a consistency model. If low latency is important, we strongly advice system architects to think about their event design and apply practices of *domain-driven-design* (DDD), such as breaking down their application to trivial facts (domain events) and derived aggregates and categorizing them as monotonic or non-monotonic, as described in subsection 2.1.8.1. Afterwards, the appropriate consistency model can be decided upon, which further guarantees safety and liveness for the adapted system design. If latency is not that important or can be mitigated in other ways through sharding and other practices, we recommend opting for strong consistency, as this ensures safety in any case (if thee are no out-of-order events) and allows starting with a simple system design that can be better thought out afterwards. Regarding out-of-order events, they must be either disallowed to provide strong consistency, or the consistency level must be explicitly flagged as eventual consistency, even if a strongly consistent replication protocol is used. We also recommend to look at time-bound partial consistency, as it can help with out-of-order events (subsection 3.3.4.4). We hope that this work will serve as a guidance here.

The Model Decision. Accordingly, we opted for strong consistency as the strongest of the identified models. It allows the event store to act as a platform that suits all use cases at least regarding safety. In addition, this allows us to built for a set of use cases where data safety is a top priority, and enables use cases where patterns that include the non-occurrence of events are to be detected. New distributed storage systems that we analyzed have proven that very high throughput is still possible in practice when a strongly consistent replication protocol is applied (see subchapter 3.2 for examples). With strong consistency, we can start with a prototypical implementation that is easy to understand, compared to other protocols. We then evaluate this implementation to understand the trade-offs for latency and availability not only in theory, but also in practice. We are aware that this introduces network latency and decreases the maximum throughput on a single stream, while this effect can be mitigated with horizontal scaling through partitioning and sharding. Many popular distributed databases have proven that extremely high throughput rates are possible even with strong consistency, as we have shown in section 3.2. This serves as a basis for future work, that could result in a distributed ChronicleDB that allows users to choose their consistency level on a per-stream basis, or even across streams. What we found so far—and this is backed by our observations of other distributed systems—is that metadata should always be distributed with strong consistency,

even if data is replicated with a lower consistency level, so that the clients and the cluster nodes themselves never assume an inconsistent cluster state, which can lead to erroneous behavior.

We identified a few strategies how to cope with out-of-order events for those clients with strong ordering requirements by event time. We decided to implement a lightweight buffer to cover the inconsistency window, while we do not implement advanced techniques due to the limited scope of this work. For clients that want to process events by insertion time, linearizability on the data perspective is sufficient to ensure consistency on the client perspective, even if the events occur out-of-order.

This considerations justify our decision that we add the following requirement to the distributed event store:

- **Linearizability**: To ensure the safety of the event streams, every write to an event stream will be ordered, i.e. every event is written in the order given by its timestamp, and every query will return the events in this order. The order can be either by event or insertion time. An expection to this can be made with out-of-order events, that must be written at least ordered in insertion time, but still returned ordered in either event or insertion time.

Trade-Offs. With strong consistency and buffered inserts, we provide a generalized solution that somehow works in every case, but not perfectly in a specific use case. This can only be achieved by allowing users to control various parameters that affect consistency. There are latency and availability trade-offs to be made. This is not necessarily a bad thing: We have learned that compromises cannot be avoided, while the effects of the compromises may not be so bad in practice. We will introduce other strategies to cope with this, such as partitioning.

The buffer can handle out-of-order arrivals of events, if its size and timeout are chosen wisely. But as this will always be a heuristic, events might still arrive out-of-order. In addition, the buffer increases the latency of write acknowledgements, as any event is only written and acknowledged once it's batch is acknowledged.

3.3.5 Deciding for a Replication Protocol

Now that we decided for strong consistency, the space of possible replication protocols (presented in detail in chapter 2) is narrowed down to a few, such as consensus protocols, state machine replication and derivates. Of these protocols, the protocol of our choice is Raft (see subchapter 2.2), or more precisely: Multi-Raft. Here follows a list of reasons that justify the use of Raft for our purposes:

- It provides strong consistency,
- An implementation can be derived from the complete and concise specification, which reduces the chance to introduce violations of correctness guarantees,
- It is a understandable protocol, which facilitates the application and customization of the protocol,
- It is commonly used by some of the largest distributed systems vendors in large-scale production systems (cf. section 3.2),
- Academic research on Raft has been trending in recent years and will likely continue to do so in the coming years, allowing for a reliable stream of updates and improvements to the protocol,
- There are many ready-to-use implementations of Raft with proper APIs written in major programming languages and supported by large communities that will also adopt improvements from recent scientific research,
- Multi-Raft allows us to implement partitioning and sharding natively,
- The monotonically growing append-only write-ahead log that contains the commands (i.e. the events) could be naturally paired with the ChronicleDB event log according to “the log is the database”,
- The log also allows to work with out-of-order events efficiently, compared to primary-copy replication,
- In the face of strong consistency, the cost of replication are justifiable in Raft.

3.3.6 Raft Implementations

There are several Raft implementations in different programming languages. The Raft authors maintain a curated list of these implementations on their website [185], where they provide an overview of the following properties and features per listing:

- GitHub stars,
- Authors,
- Programming language,
- License,
- Leader election + log replication,
- Persistence,
- Membership changes,
- Log compaction.

There may be a few other libraries that are not on this list, but we haven’t investigated that yet because this list looks very comprehensive.

3.3.6.1 Library Decision

Before introducing the library we have chosen, we list the requirements we have for a suitable library implementation.

Requirements. We are looking for a library written in `Java`, since ChronicleDB is written in `Java` and we want to integrate it tightly with the actual event store implementation, rather than offering it as a service alongside ChronicleDB (cf. ZooKeeper).

We are also looking for

- A library with a non copyleft open-source license,
- A library that is likely to be maintained in a future,
- A library that is feature-complete in the sense of the original Raft protocol description: it provides leader election, membership changes, log replication and persistence, handles network reconfigurations, and allows for log compaction,
- A library that looks to fulfill all correctness properties.

Apache Ratis. We went for *Apache Ratis* [186] [187] since it satisfies all our requirements and more:

- It is written in `Java`,
- It can be used as an embedded library rather than a dedicated service,
- It is feature-complete,
- It allows to implement both the state machine and the log yourself, but also provides basic implementations for a quick start,
- It uses `gRPC` (`HTTP/2`) as the messaging protocol both under the hood and for replication messaging,
- It is used in a production-ready, high-throughput, data-intense implementation (Apache Ozone [188]),
- It is offered under a non-copyleft open-source license (Apache 2.0),
- The project is relatively popular on GitHub[9],
- The team continously works on the library,
- And the project was moved into the Apache Software Foundation after it succeeded in the Apache Incubator.

Ratis comes of course with some caveats:

[9] GitHub stars are a good indicator for the popularity of a library and a proxy for the future-safety, as more popular libraries are more likely to be maintained in future.

- The library is not well documented (a large part of the documentation is not only available in English),
- Due to the not-so-good documentation, parts of the API are hard to understand and the learning curve is steep (can best be learned by examining their sample projects, Apache Ozone implementation, and interfaces),
- The backlogs and current project progress are hard to understand from what they list in their JIRA project[10],
- The team does not release updates very often (less than have a year) [11],
- There is no (automatic) verification of the Raft correctness properties.

Unfortunately, we haven't found any `Java` Library with such a correctness verification. We compared its source code to the TLA$^+$ specification of Raft and it looks quite valid, while we can not exclude any exceptions in the runtime behavior.

Differences to Basic Raft. The Ratis authors claim that Ratis is not only suitable for metadata replication, but it is actually targeted to high-throughput data replication problems. We have seen in subchapter 3.2 that some vendors, such as InfluxDB, do not replicate their data using Raft, but only the meta data, because of the effect it has on the throughput. In theory, their data replication is only eventually consistent, while they claim this is not the case in practice under normal throughput conditions. This is similar to our observations of the inconsistency window (see section 3.3.4.4). With this in mind, Ratis seems to solve a major problem and makes it suitable for our purposes.

Following is how the authors describe their approach [189]:

While in basic Raft, all transactions and the data are written in the log, which is not suitable for data-intensive applications, in Ratis, applications could choose to not write all the data to log, by allowing engineers to manage state machine data and log data separately.

To do so, they provide an interface `StateMachine.DataApi` for managing data outside the Raft log. This allows to build a very light-weighted Raft log.

We haven't implemented this yet in our prototype, but sketched out how this could be achieved in paragraph 3.3.7.6.

Ratis also introduces the concept of *divisions*. A division represents the role a node has in a particular Raft group. In Multi-Raft, a node is allowed to take part in multiple Raft groups, thus having multiple different roles at the same time.

[10] https://issues.apache.org/jira/projects/RATIS/issues

[11] We are using Ratis in version `2.1.0`; as the time of this writing, there is a version `2.3.0` released.

Alternatives. Other alternatives are listed as follows and why we didn't choose them.

- **Hazelcast**: Hazelcast is a full-fledged platform for building real-time applications with stream processing capabilities, that also uses Raft under the hood. This sounds very suitable for our case, as this is exactly what we want to build. But Hazelcast is not a embeddable library, but a platform with a magnitude of other features (in-memory key-value store, embedded stream and batch processing, pub/sub messaging). It is available as an open-source edition with community support, but you still build your application on top of the Hazelcast platform, instead of integrating Hazelcast. This introduces additional complexity, especially as we want to build ChronicleDB as a platform itself, which would require a lot of re-engineering of Hazelcast. Hazelcast is therefore aimed at application engineers rather than platform engineers. To embed Hazelcast instead of running it as a platform, you need to register yourself as a commercial partner of the Hazelcast, Inc. company.
- **etcd/raft**: The most popular Raft implementation out there. As it is written in *Go* (aka. Golang), it is not suitable for our case (only if we build replication in ChronicleDB around a microservice architecture, which is not our goal). We list it here for the sake of completeness. etcd/raft powers etcd, a high-throughput key-value store, which itself is the fundamental basis for Kubernetes.

We use `Java` for simplicity, as ChronicleDB is written in `Java`. For best performance and future-proof support (at the time of this writing), it is recommended to implement the library yourself or to use one of the popular Golang implementations: etcd/raft or hashicorp/raft. One could even use the *Gorum framework* in Golang—with some adjustments as in [190]. Go is a relatively new language that is very popular in the realm of distributed systems, since it provides a powerful yet easy built-in networking library similar to Erlang, and the language semantics are simple which makes it easier to write, read and maintain distributed algorithms in Go, compared to other languages. Another reason is the already broad support in this area from package providers, which makes it easier to get started.

Another alternative we have considered is to write the replication layer ourselves from scratch. This would have allowed us to implement a replication layer that allows the consumer of the event store to decide on the level of consistency. We have not found a library that allows this, so this must always be implemented by yourself. As we have seen so far, there is no out-of-the-box implementation of multiple protocols that can just be toggled in between. We found the approach in

Apache IoT DB to switch the protocol by wrapping it in a provider, but only for Raft, multi-leader consensus and standalone use[12]. Another approach would be to limit the toggleability to a per-stream basis and deploy two completely different replication protocols or libraries. Due to the sheer complexity of such an undertaking, we quickly dismissed the idea.

Note that in all of the previous implementations we presented in subchapter 3.2, the replication layer was built by the engineering teams themselves.

Now that we presented the library we use for our prototype implementation, we will present the implementation itself in the following sections.

3.3.7 ChronicleDB on a Raft

This subsection describes the actual implementation of Raft in ChronicleDB with Apache Ratis. Before we delve deeper into this, let's look at further limitations.

3.3.7.1 Limitations

On top of the constraints we set for the research (cf. section 3.3.2), we also limit the scope of our prototype implementation to fit within the bounds of this work. For topics not covered here, see also the section on future work in Chapter 5.

Sharding. We limit our implementation to the one-partition-per-stream case, thus we are not able to evaluate the impact of sharding on the overall throughput.

Queries. To allow for any queries, the original query interface of the ChronicleDBs `lambda-engine` must be adjusted, since it exposes a cursor interface that is hard to break down and wrap in a serialized message, which is required to be executed by a Raft node. We limit our work on implementing and investigating writes, since reads are served from a single leader and do not incorporate significantly more network round-trips, thus not being slower than in the standalone deployment—in fact, replication can reduce read latency. This is sufficient for our analysis. Since it was much easier to implement basic aggregate queries, we use them to validate our results.

Message Protocol. We use gRPC with `Protocol Buffers` for intra-cluster messaging between nodes. But, we currently only offer `HTTP/1.1` for clients to communicate with a deployment of ChronicleDB (next to the `Java` API for the embedded case).

[12] https://github.com/apache/iotdb/tree/master/consensus/src/main/java/org/apache/iotdb/consensus

We recommend using gRPC (which is built upon HTTP/2), AMQP, or other protocols that are best suited for high-velocity event messaging.

Advanced Out-Of-Order Handling. While we recommend to invest time into researching and implementing an approach to cope with out-of-order events such as punctuation, time-bound partial consistency with inconsistency windows, or a consistency query module, the scope of this work is limited. Hence, our prototype implementation does not provide such a module. Instead, we will implement a lightweight buffering approach for writes that dramatically reduces the likelihood of inconsistencies due to out-of-order events in practice.

Log Optimization. I/O on the Raft log can be costly, as all operations must be written to the log before executed on the event store replicas. Without buffering, the log looks almost similar to the actual event stream persisted in the store. In addition, it requires non-trivial state management such as allocation and pruning to prevent unbounded growth.

The log implementation in this work is far from being optimized. Even more, we violate the "The log is the database" philosophy of ChronicleDB. Ratis allows us to implement the state machine and Raft log separately, and allows for a lightweight log implementation to enable high-throughput data replication. There are also approaches for log-less state machine replication in the literature that can lead back to this philosophy [95]. But, we limit our work on the replication protocol itself, not on the implementation details of the log. We sketch out a possible log improvement in paragraph 3.3.7.6.

Log Compaction. We haven't implemented a snapshotting mechanism so far. Currently, when the system is rebooted, the whole log is replayed and the TAB$^+$ index rebuilt, which is very expensive. While snapshotting a simple key-value store is simple (which previous literature was limited to), snapshotting the event store by creating clones of it to the current point in time is too expensive, as the store is already written to disk of the replicas. We considered that it is sufficient to reduce the snapshot to a pointer that points to the latest committed event in the store, and changing the state transfer implementation of Ratis to one that transfers the TAB$^+$ up to this pointer. But this method may be limited to an append-only event store—we haven't considered a solution for out-of-order entries yet, nor are we sure that this requires expectional handling at all.

3.3.7.2 Technical Architecture Overview

This subsection outlines the architecture of ChronicleDB on a Raft. The system is implemented in 3 layers, as illustrated in figure 3.23:

- **Replication Layer**: This layer is built with Apache Ratis. We use the `Java` API of Apache Ratis to implement a Multi-Raft cluster running a distributed ChronicleDB state machine. This layer is described in subsections 3.3.7.4—3.3.7.9.
- **Messaging Layer**: In this layer, messaging between Raft nodes is defined with Google `Protocol Buffers` and sent over `gRPC`. We elaborate the details of this layer in subsection 3.3.7.8. In addition, there is also messaging between ChronicleDB and clients, which is currently solved with `REST` on `HTTP/1.1` (cf. subsection 3.3.7.1).
- **Data Layer**: This is the layer that holds the actual ChronicleDB implementation by wrapping the embedded ChronicleDB `EventStore` implementation and providing an implementation of the `ChronicleEngine` as the API for producers and consumers. We present our changes in this layer in subsection 3.3.7.3.

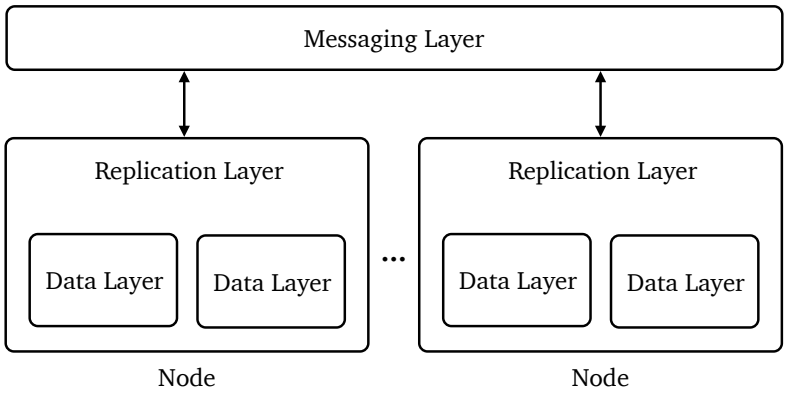


Figure 3.23: The 3 layer-architecture of ChronicleDB on a Raft. Each nodes runs on a replication layer and communicates with each other over the messaging layer, while they manage multiple data layers (event streams).

3.3.7.3 Changes in the ChronicleDB Core

A few adjustments where to be made on the original implementation of ChronicleDB. Our adjustments are made on the basis of ChronicleDB version `0.2.0-prealpha`. The original implementation consists of multiple dependent packages:

- `chronicledb-event`: The ChronicleDB event store, which is dependent on the following:
  - `chronicledb-index`: The indexing layer of ChronicleDB,
    - `chronicledb-common`: Commons for the ChronicleDB packages,
    - `chronicledb-io`: The I/O layer of ChronicleDB,
  - `event-data-model` from `de.umr.event`: The event meta model, interface and utilities to describe an event schema.

With this setup, ChronicleDB can be deloyed embedded as a library in other `Java` applications.

On top of this exists the `lambda-engine`, which is a Spring Boot app providing both real-time query and batch processing APIs, based on the Lambda architecture. In addition, it allows to deploy ChronicleDB standalone rather then embedded, exposing a `REST` API for clients to insert and query events (and aggregates).

Our prototypical implementation is dependent on the following packages:

- `chronicledb-event` as the ChronicleDB core implementation,
- `event-data-model` as we need the event data model across all layers.

We wrap the embedded ChronicleDB library into the data layer of our implementation and re-engineered parts of the `lambda-engine` to serve our implementation in a test application (cf. section 3.3.8). We created an interface to the `ChronicleEngine`, the centerpiece of the service, and extended it to a replicated one, serving as a facade and the API for clients. We illustrated this in figure 3.24. Since we wrapped the relevant classes and interfaces of ChronicleDB into own extensions and facades, no changes in the core implementation of ChronicleDB where needed. To allow for distributed queries, the original query interface of the `lambda-engine` should be adjusted, since it exposes a cursor interface that is hard to wrap in a serialized `gRPC` command message. Instead of wrapping, it should be re-engineered for the distributed case. We limit our work to support replicated writes, so we did not touch this yet.

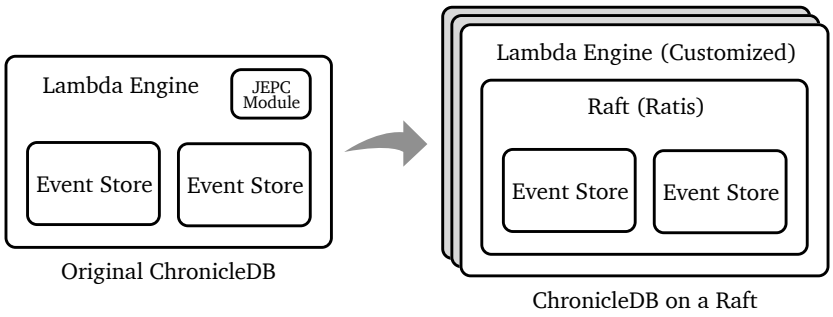


Figure 3.24: Re-engineered ChronicleDB on Raft. The replication layer sits between the event store instances and the lambda engine.

Note that the event data model in ChronicleDB is slightly different to our definition in section 3.3.4: In ChronicleDB, an event contains two timestamps t_1 and t_2, denoting the start and end of the *temporal validity* of the event. This makes discussions about the event order somewhat more complex, but we won't go into it.

The full picture of the architecture is shown in figure 3.25. The different parts of this architecture will be explained in the following sections.

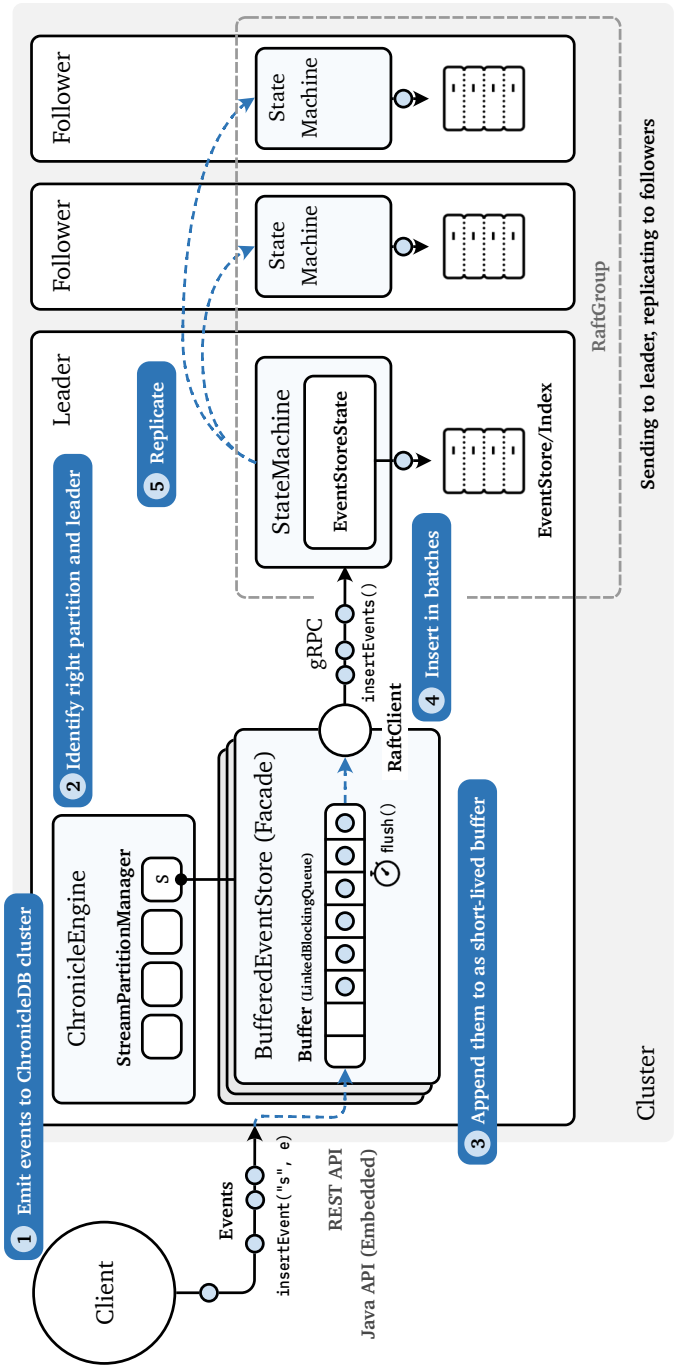


Figure 3.25: Architecture of a ChronicleDB cluster for high availability

3.3.7.4 Cluster Management

In the center of the replicated ChronicleDB implementation sits the cluster management. The cluster management is crucial for managing nodes, partitions and instances of state machines in the cluster.

The cluster manager acts as both a book keeper and as a service for provisioning nodes to new or initial partitions of a state machine instance. It

- Keeps references to all nodes, Raft groups, divisions, and state machine instances,
- Stores all metadata of the cluster, including information on the machines hardware, OS and load,
- Performs regular health checks using heartbeats (including failure detection),
- Serves Raft nodes with health info about the cluster,
- Provisions new partitions, such as for newly instantiated event streams, with load balancing,
- Acts as a gateway and routes requests to the Raft group of the according state machine and partition,
- Updates the *bootstrap lists* of the clients (the list of known nodes of a cluster a client can message to),
- Acts as a service for clients to query the cluster state.

It therefore plays a crucial part in the architecture; without it, no write or read can happen at any node.

The cluster manager is implemented as a replicated state machine itself, as it must provide at least the same level of fault-tolerance and availability as the actual application logic due to its crucial role in the architecture. The cluster management and actual application logic is divided into two instances of Raft servers, as shown in figure 3.26. This guarantees that the management server is always up, even in case of faults of the application servers, which are detected by the management server. While the management Raft group runs on all nodes, ensuring an additional level of fault-tolerance, the actual application groups run only on the nodes with partitions covering them. Moreover, the cluster manager runs another state machine that serves a simple in-memory key-value store for additional metadata that we haven't yet implemented with type safety. This key-value store allows for single-nested key-value pairs. We use this to store and distribute additional node metadata, such as remaining disk capacity, which is exposed to the user via an API and a UI and can be used for alerting or auto-scaling.

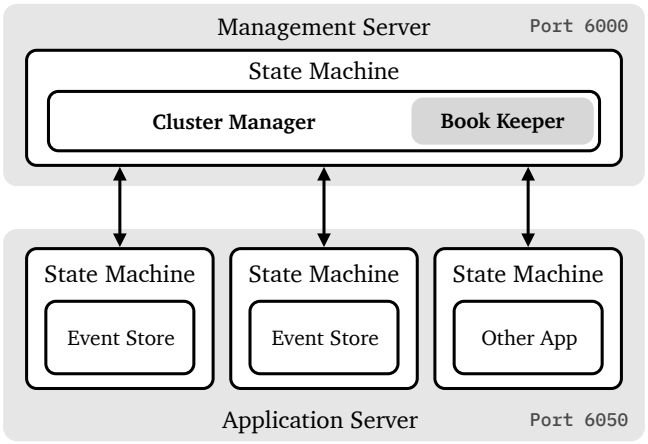


Figure 3.26: A single deployment contains both a management and application server. The management server is responsible for all book keeping of the cluster.

In Apache Ratis, every instance of a `RaftServer` runs on its own port. Therefore, we need to provide two ports per node, one for messaging and replication of the application logic, and one for the cluster management[13]. While this adds complexity for cluster operators, it ensures that both servers are decoupled. This is similar to what other vendors are doing (InfluxDB, Apache IoTDB, see subchapter 3.2). Although one could imagine running cluster management and application logic on separate machines, we run them on the same machines for efficiency and complexity reasons. This is illustrated in figure 3.38. In addition, there is one public port for client requests. In our example setup on AWS, as well as in our local Docker setup, we made both ports for cluster management and application replication private. We recommend this to reduce the risk of someone messing up the protocol messaging, and thereby the consistency of the system.

The Raft log of the cluster management state machine does not persist every command, simply to avoid additional I/O and log compaction complexity for very frequent commands such as heartbeats.

Raft Groups, Clients and Servers. Ratis provides three classes that are mandatory to run a cluster on Raft.

A *Raft group* is a logical instance, containing a set of nodes that replicate a single state machine instance. Each physical node can participate in multiple Raft groups,

[13]The literature commonly refers to a node partaking in cluster management as a *meta node*, since it manages, stores, and replicates the metadata of the system, in contrast to the *data nodes*, which replicate the actual payload data. A physical node can have both the role of a meta and data node at the same time.

meaning it can take on different roles at the same time. This relation is called a *division*. A division is a relational tuple describing a node, a Raft group, and the role of this node in this group.

A *Raft client*[14] is a client instance that can send requests to a single Raft group. On instantiation, the Raft client is given the Raft group ID and a bootstrap list of nodes to reach out to. Once a Raft client reached out to a single node, it receives information about the whole group and what the current leader is, and communicates directly to the leader from this point. A Raft client is mandatory for sending any message to a Raft group. A write request sent from a Raft client is written and replicated to Raft logs of a quorum of the nodes in the group, before it is applied to the state machine. Raft clients can be run on any `Java` service on any machine as long as they can reach one Raft server of a Raft group and know the ID of the Raft group to connect to.

A *Raft server* is the instantiating of Raft on a single node. Ratis supports Multi-Raft, so a Raft server can run multiple Raft groups.

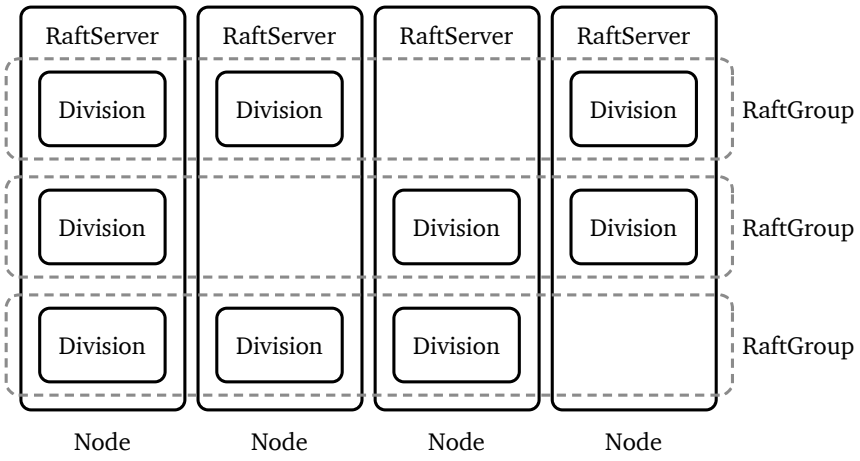


Figure 3.27: Multi-Raft in Apache Ratis, here with a replication factor of 3

Commands. As we describe in subsection 3.3.7.8, all communication between nodes is done via `gRPC`. All possible commands and message schemas are defined in `Protocol Buffer` schema descriptions. This provides a clear, transparent and reliable interface for all communication with the cluster management server. The

[14]The Ratis Raft client implements some of the client interaction semantics presented in the Raft dissertation (cf. section 2.2.12), but due to a lack in consistent documentation of Ratis, we could not find out which.

server supports the following commands:

- `REGISTER_PARTITION(string statemachine_classname, string partition_name, int replication_factor)` to register a new partition for a set of nodes and a state machine with the given replication factor,
- `DETACH_PARTITION(string partition_name)` which detaches a partition without deleting its persisted contents,
- `LIST_PARTITIONS(string statemachine_classname)` to list all registered partitions, either for all or a specific state machine,
- `HEARTBEAT(RaftPeerProto peer)` for a node (*peer*) to broadcast a heartbeat to let the cluster know of its presence.

In addition, the metadata state machine provides the following commands:

- `SET(string scope_id, string key, string value)` writes a value to a key for a certain scope. If the key is already defined, it is overwritten. The scope allows to define multiple instances of the key-value store, identified by the scope ID.
- `DELETE(string scope_id, string key)` removes the key from the scope, if existing,
- `GET(string scope_id, string key)` returns the value for the key of the given scope,
- `GET_ALL_FOR_SCOPE(string scope_id)` returns the whole map for a scope,
- `GET_ALL()` returns all maps for all scopes.

3.3.7.5 Event Store State Machine

The ChronicleDB event store is split in multiple event store instances, where each instance manages a particular event stream for a given event schema. In a standalone ChronicleDB deployment, these event store instances are managed by the `lambda-engine`, which maintains a map of references to the respective instances. In the basic approach, the list of available stream is managed with a metadata file on disk, next to the event streams. Each event stream itself is stored a set of files, which contain the block segments, the index, and secondary indexes, if any.

In the distributed approach of this work, the event store instances are managed by the cluster manager, which keeps book of all the registered event store instances as partitions. Each ChronicleDB event store instance is managed by a dedicated state machine instance. In this subsection, we examine our implementation and additional

abstractions that we have built on top of Apache Ratis to improve maintainability and generalizability.

Formal Definition. In its core, the state machine for an event stream can be formally described as follows (based on subsection 2.1.9.2), under the condition that the only valid write operation is the insertion of one or multiple events:

C is the set of all possible events e.

$S = \{\dots\}$ denotes all possible sequences E of events.

Writing one event can be denoted as

$\delta(e, \langle\rangle) = \langle e \rangle$ where $\langle\rangle$ is the empty stream.

For non-empty streams:

$$\delta(e, \langle e_1, \dots, e_n \rangle) = \begin{cases} \langle e_1, \dots, e_n, e \rangle & \text{if } e \geq e_n, \\ \langle \delta(e, \langle e_1, \dots, e_{n-1} \rangle), e_n \rangle & \text{else (out-of-order).} \end{cases}$$

and writing events in batches can be described by

$$\delta((e'_1, \dots, e'_m), \langle e_1, \dots, e_n \rangle) = \delta(e'_m, \dots \delta(e'_1, \langle e_1, \dots, e_n \rangle) \dots)$$

Generic State Machine Instantiation. Our architecture allows us to run not only instances of event stores, but instances for any arbitrary implementation of a state machine. This allows us to break down the replicated logic into multiple state machines for different service domains, which improves the performance of a complex system since this approach allows us to break coordination and linearization of operations where we don't need it. However, we must be careful not to decouple logic with a great amount of strong causal connections. We can either deploy the state machines from the same code base or seperately in a service-oriented approach. Both are valid approaches since the communication between different state machine instances is abstracted using Raft clients and Raft groups. They handle to build a connection with the appropriate group leader via `gRPC`, while the messaging between groups is reliable due to the downward and backward compatibility of message schema evolution in `protobuf`. We illustrated this in figure 3.28. It is also imaginable to introduce a message bus, i.e. using Kafka (or ChronicleDB itself) to decouple communication between state machine instances.

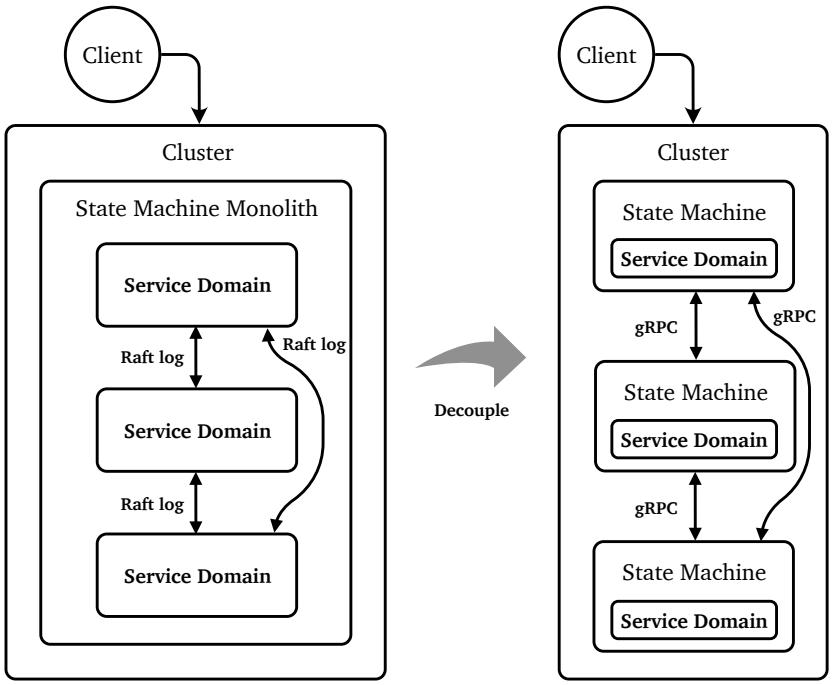


Figure 3.28: Decoupling multiple service domains from a monolithic state machine that must not neccessarily be linearized into multiple state machine instances can improve the performance of the overall system

Thanks to Multi-Raft in Ratis, each Raft group can run a different state machine implementation. A fresh state machine instance is registered as a single partition via the cluster management. You can instantiate such a instance using the `StateMachineProvider` interface. It bundles the state machine to be deployed, a descriptor for that instance/partition, the number of replicas to partake in that partition, and other domain-specific information such as the event schema. When issuing the creation of a new event stream for a given schema, the API creates such a provider and registers it to the cluster management, which then executes the provisioning by selecting the replicas with the lowest load, spawning a new Raft group for the requested state machine instance and writing it to the cluster managers log.

There is one state machine instance for each event store instance, which covers a single event stream. Currently, there is exactly one partition per state machine instance, as we don't support sharding yet. Our implementation of the `lambda-engine`

makes use of the cluster manager to retrieve the set of instantiated event store state machines, so it can forward client requests for writes and queries to the respective event stream.

The State Machine Interface. Our state machine interface abstracts the original Ratis interface in multiple layers to improve the maintainability of the code and the system and to allow to implement new state machines with less effort and highest possible modularity.

In Ratis, the state machine API entices you to build a monolithic state machine, which quickly becomes hard to read and to maintain. With our approach, we enforce modularity and atomicity. Each abstraction layer of the state machine has a specific set of responsibilities (the layers are illustrated in figure 3.29 as well as in the UML diagram in figure 3.31):

- The abstract `ExecutableMessageStateMachine` class can be extended to provide lightweight state machines that uses `ExecutableMessages` together with `OperationExecutors` to decouple state and operations on the state. It simply references the `State` that is to be managed, the `StateManager`to manage the state, and the class of allowed `ExecutableMessages` on this state.
- The `StateManager` interface must be implemented to keep a reference on the `State`. Direct manipulations on the `State` object are prohibited, instead all allowed operations must be implemented via the `StateManager`. The `StateManager` is passed to `OperationExecutors`, exposing its interface for atomic state updates.
- The `State` interface should be implemented to describe how the state should be instantiated (what the empty state looks like), how it is persisted (references to the actual storage, such as the node's disk) and how snapshots are taken (for log compaction).

In Ratis, a state machine has at least two methods `applyTransaction` and `query`.

- `applyTransaction` receives the `TransactionContext`, which includes the Raft log entry. The log entry itself includes the message payload for the execution on the state. Developers are responsible to update the last applied term-index themselves in case the operation was successful, otherwhise to rollback the state change (in case it is not atomic).
- `query` is similar to `applyTransaction` with the difference that it does not

receive a log entry, but only the message payload. Query operations are not allowed to mutate the state, therefore, no entry is written to the Raft log.

We generalized the `applyTransaction` method. Developers do not longer need to update the term-index themselves. Instead, all operations must be written atomically asa pair of an `ExecutableMessage` together with an `OperationExecutor`. For the case of a failed execution, every `OperationExecutor` allows to provide a `cancel` method for neccessary rollbacks or compensations, in case an operation can not be designed atomically and/or idempotent.

For side effects (not on the state—this is prohibited—but rather for logging or notifications to the cluster manager) we added a `beforeApplyTransaction` respectively `afterApplyTransaction` method.

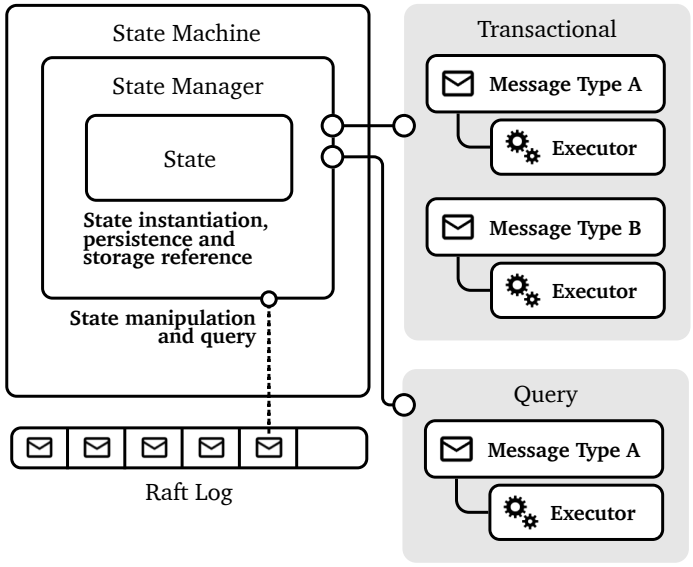


Figure 3.29: Splitting up the state machine interface into the state machine, the state manager, the state itself and message-executor pairs.

The Event Store Facade. The actual event store instances that manage the event streams are prevented from direct access. They can only be accessed through the replication layer, which ensures consistency. For convenience, an event store facade hides the replication layer and offers the same API to producers and consumers as the original event store classes, since it implements the same interface. This follows the philosophy of strong consistency to render a replicated system to clients as it where a single-server system. We extended this approach to the `lambda-engine`,

which offers an API to access multiple streams. However, we haven't managed to implement neither continuous queries nor batch processing yet (cf. subsection 3.3.7.1).

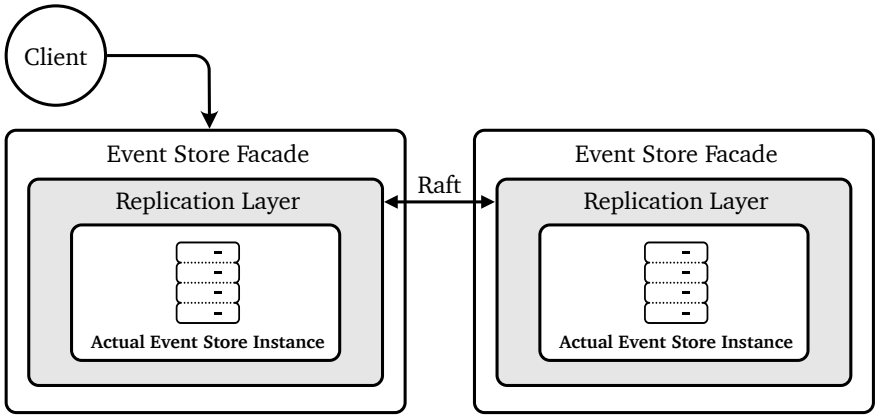


Figure 3.30: The actual event store instances are hidden behind a facade. The replication layer wraps the event store instances, preventing direct access to them.

Architecture and Message Flow. We illustrate the class hierarchy and the message flow in the annotated UML diagram in figure 3.31.

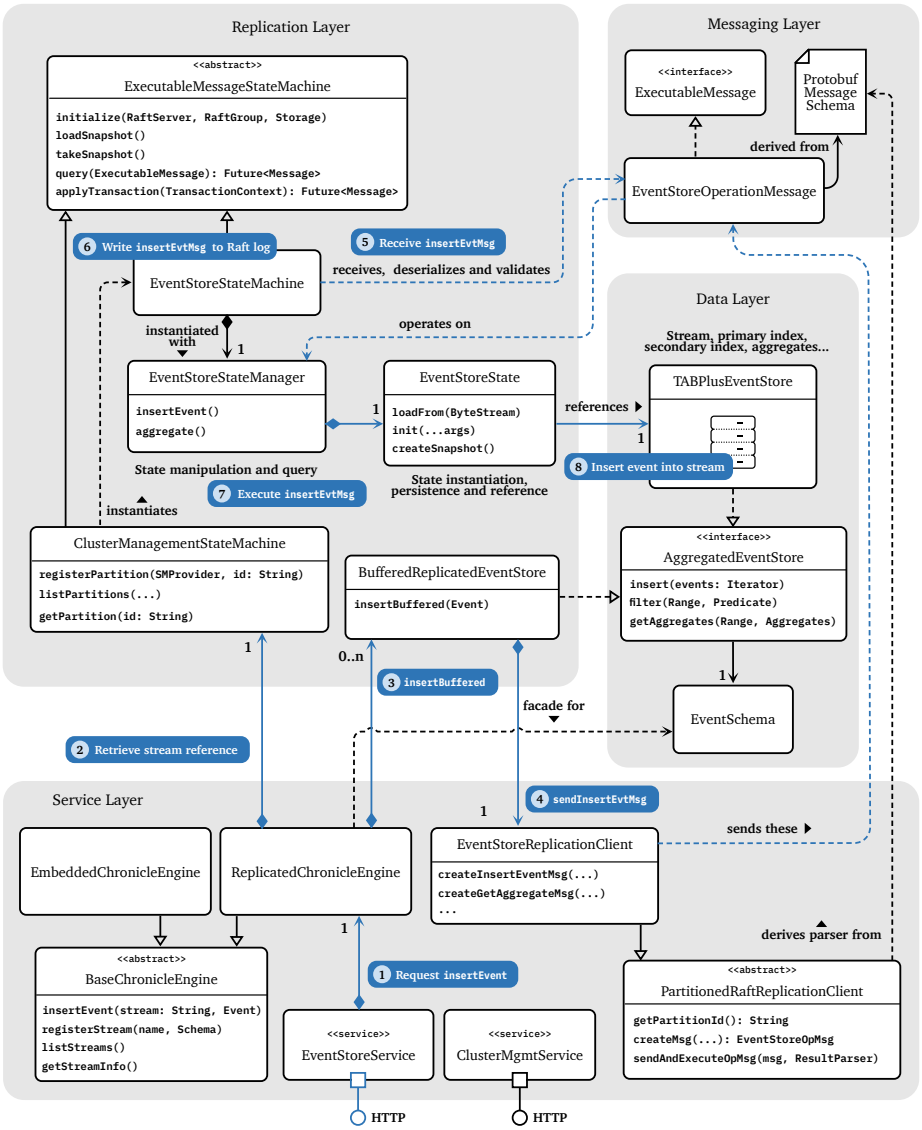


Figure 3.31: UML class diagram for the replicated ChronicleDB event store state machine. The annotations show the message flow for an event insert operation. A non-annotated version can be found in the appendix.

An insert operation includes the following components in several steps:

(1) The `insert` of an event is requested via the HTTP API.

(2) The reference to the target stream (partition) is retrieved from the cluster management. It is necessary to use it to instantiate the Raft client so that it

sends the request to the leader of the coresponding Raft group.

(3) The insert operation of the `BufferedReplicatedEventStore`is called, which is a facade for the Raft client. It implements the same interface as the actual event store and additionally provides a buffered insert operation.

(4) Once the buffer is flushed (meanwhile, other insertions could have been requested), a `InsertBulkEventsOperationMessage` is instantiated. Its payload contains the events to be inserted from the buffer. Before flushing, the events are ordered by event time. The message is then serialized using `Protocol Buffers` and sent via `gRPC` to the leader of the Raft group that manages the streams partition.

(5) The leader receives the message...

(6) ...and writes it to the Raft log. The log entry is now replicated to a quorum.

(7) Once the message is replicated to a quorum, the `EventStoreStateMachine` instances at each quorum node (and successive other nodes) apply the message, executing the `insert` command.

(8) The `insert` operation is finally executed on the event stream by the actual event store instance on all participating and available nodes.

Note that in future work, the order of this operations may differ if the Raft log is optimized; i.e. a message is applied immediately on the state machine while writing a pointer reference only to the Raft log (cf. section 3.3.7.6).

Commands. The prototype implementation of the event store state machine supports the following commands on a stream:

- `PUSH_EVENTS(repeated<Event> events, bool is_ordered)` inserts the given list of events, which is optimized by telling ChronicleDB in advance if these events need to be reordered before inserting,
- `AGGREGATE(Range range, AggregateScope aggregate_scope, AggregateType aggregate_type, string attribute)` derives an aggregate of the given type (currently supporting `COUNT`, `SUM`, `MIN` and `MAX`), either on a attribute or global scope, for the given time range,
- `GET_KEY_RANGE()` returns the time range for the stream,
- `CLEAR()` returns the whole stream.

A query message is not yet implemented due to the complexity of the query interface and more challenges that we pointed out in subsections 3.3.2 and 3.3.7.1.

3.3.7.6 Log Implementation

The Raft log in Apache Ratis has all the properties that a Raft log must have: it is replayable, compactable and comes with a term-index pair for each log entry to mark the current leader term and the index of this entry for linearizability. It also denotes the high-water mark of the log with the `lastAppliedTermIndex`. In addition, Apache Ratis allows to implement the Raft log yourself, and independently from the state machine, by providing an interface for managing log data outside the Raft log. For data-intense applications, we recommend to do so: they provide a basic, non-optimized Raft log implementation that dumps the log entries to segmented binary files on disk, and reads them again from disk when feeding them to the state machines. This basic implementation is not efficient, slowing down data-intensive applications. In our evaluation, writing every event to the Raft log slows down the system tremendously, due to increased I/O and especially because the default log in Ratis requires a log entry to be written before it is sent to other nodes, effectively blocking the whole system on every single operation (this could be mitigated by parallelizing log writes and replication). As a key insight, the Raft log implementation is crucial for the performance of the whole system.

Some of the applications we investigated use efficient embedded storage engines such as *RocksDB* that are optimized for high-throughput[15], some others wrap the log in a cache[16], and some even come with additional WAL logs, providing a multi-layer architecture of the Raft log. Even if this comes with some issues, we can learn from it to implement a better log. We use buffering to improve the performance of the system, avoiding too many writes to the log while at the same time sacrificing atomicity of the log entries. This approach is presented in subsection 3.3.7.7. In the next paragraph, we also present an approach to the log that we haven't managed to implement in our prototype yet.

[15] CockroachDB uses RocksDB for their log: https://github.com/cockroachdb/cockroach/issues/38322

[16] Hashicorp Raft wraps a cache with a in-memory ring buffer around the Raft log to avoid I/O on recently written log entries, as the state machine, replication engine and log disk writer can immediately pick up the log entry from the buffer after an insert: https://github.com/hashicorp/raft/blob/main/log_cache.go

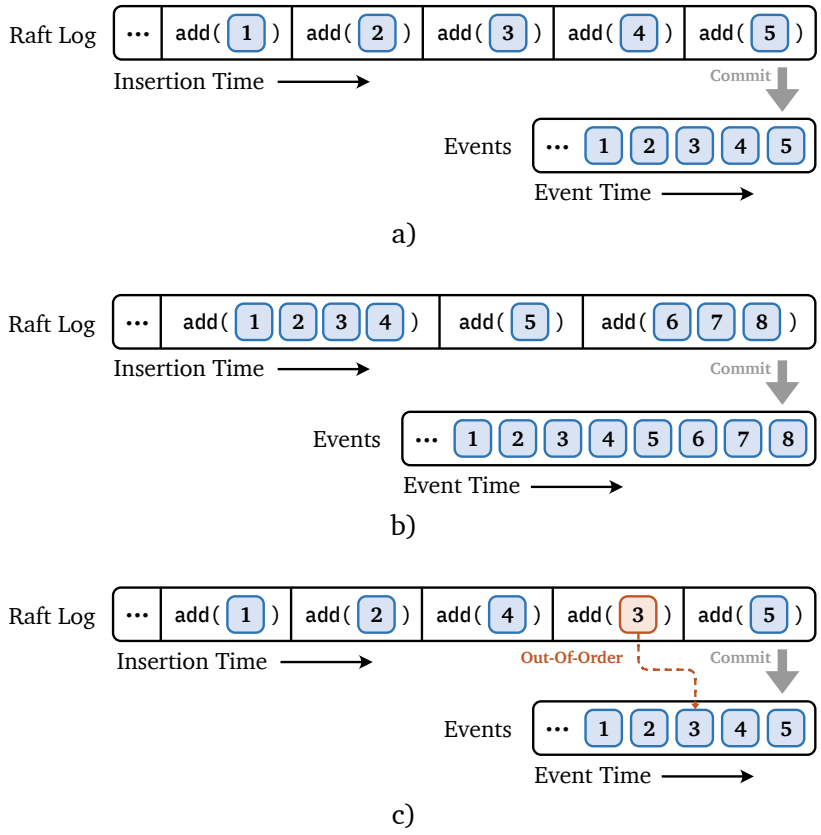


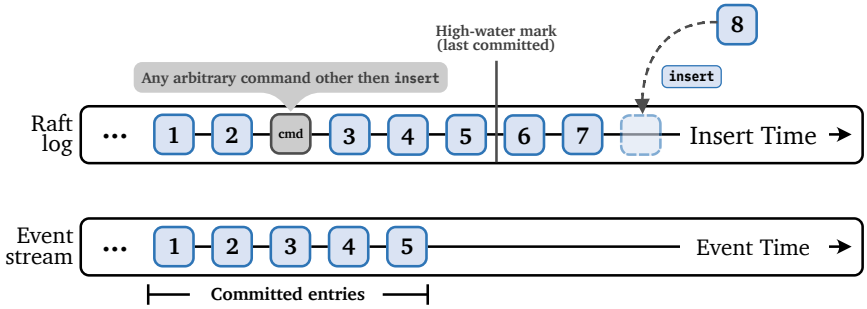
Figure 3.32: Relation of the Raft log and the event store. a) Appending events without allowing for out-of-order events always results in the same sequential order of the events in the Raft log (wrapped in operations) and the events in the event store, even if insertion and event time differ. b) With buffered inserts and without out-of-order insertions, the order correlation still applies, as the buffer contents are ordered before insertion and the buffer can only be served from a single leader node. c) As soon as out-of-order insertions are allowed, the order of the two logs given by insertion and event time is no longer guaranteed to correlate. This can break previously derived non-monotonic aggregates, as shown in figure 3.12.

Log-Less Rafting. With the Raft log, we write the same event twice: Once to the Raft log, and once to the event stream (when it is replicated and committed). The messages end up being stored redundantly, and the duplicate writes use up additional I/O capacity, which slows down the system. It also violates the *"The log is the database"* philosophy of ChronicleDB. So the question is: why can't the event stream just serve the log instead of creating a redundant Raft log?

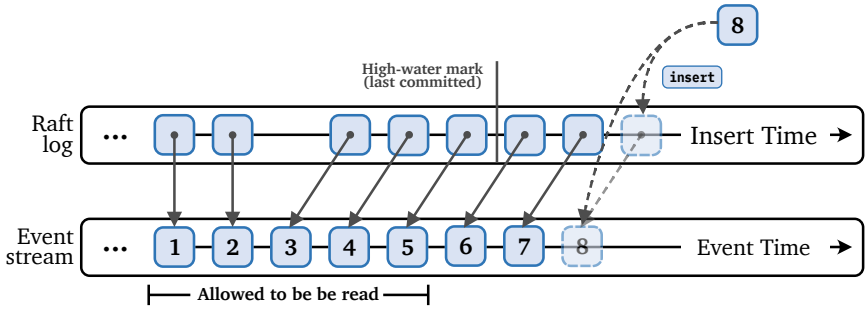
- The Raft log contains not only events (actually insert commands), but also Raft commands (such as leader changes, network reconfigurations etc.) and in

the future additional operations on the event stream (for example for schema evolution, but here we would recommend to just create a new stream).
- Not all entries in the log are already committed and applied to the state machine, similar to a WAL.
- With out-of-order events, the ordering of the Raft log and event stream can differ (see the previous discussions in section 3.3.4).

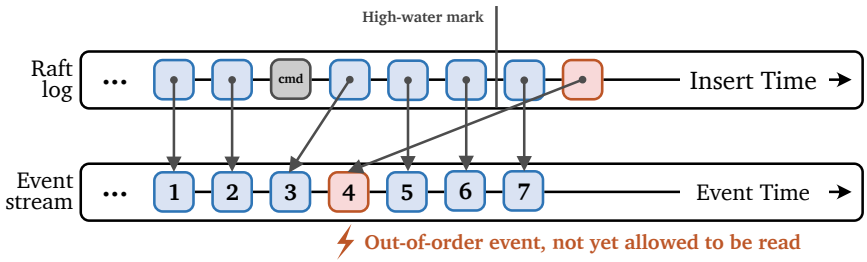
As we wrote, other applications use embedded storage engines for high-throughput to write the log. ChronicleDB is such a high-throughput embedded storage engine. Consequently, this leads to the consideration of using the ChronicleDB event store itself as the log. We can address this by reducing the Raft log to pointers, at least for the write operations. We use the event stream as the truth for the log and keep only pointers in the Raft log (next to the command, in this case `insert`). Once a message arrives at a node, the state machine immediately writes it into the event stream and at the same time a pointer to the Raft log. When reading the event stream, the system must read the high-water mark of the log to know which event stream entries are already committed, and is only allowed to return these. In case of uncommitted entries dropped from the log (e.g., after a network partitioning or due to some faults), the event stream entries after the high watermark are allowed to be overwritten. Overwriting entries is a new concept to ChronicleDB that must also be discussed from an index and storage layout perspective, therefore we haven't implemented this throughout this work. This approach must also take care of out-of-order events: it is not sufficient to read the high-water mark and allow to return all past entries in the event stream that have an event timestamp before the watermark, because this could return out-of-order entries that were not yet fully replicated to a quorum and committed. It is only allowed to return entries with an insertion timestamp before the entry with the high-water mark, which requires further thoughtful engineering to be efficient. In addition, no materialized aggregates are allowed to be returned that are based on uncommitted out-of-order entries. This is illustrated in figure 3.33.



a) Regular Raft log



b) (Almost) logless Raft



c) (Almost) logless Raft with out-of-order events

Figure 3.33: Sketch of an almost logless Raft. a) Regular Raft log, with events only written to the stream once committed, but then, they are written twice. b) Almost logless Raft, where the Raft log only contains pointers to the actual events in the stream. Events are immediately written to the stream, but only those up to the high-water mark are allowed to be read by clients. c) This simple approach won't work if out-of-order events come into play.

Ignoring out-of-order events, snapshotting the state and compacting the log is also easy, because it will mean no more than just removing old pointers that we are not interested in and keeping the high-water mark as the pointer to the most recent

stream event.

There are also more approaches to logless state machine replication in the literature, that goes way beyond this. So far, they are only suited to smaller key-value stores, as they manage a full sequence of states, but research here could be deepened for larger structures such as an event store [95].

3.3.7.7 Buffered Inserts

In his dissertion on Raft [88], Ongaro describes batching and pipelining support for Raft log entries and explains their importance to system performance. Collecting and executing multiple requests (Raft log entries) in batches reduces I/O and network round-trips at a high level. Unfortunately, Ratis does not support batching itself. Therefore, we implemented a buffer ourselves to provide batching. In this subsection, we carry out implementation details, while we refer to subsection 3.3.4.5 for more details.

One caveat of our implementation is, that log entries lose atomicity: instead of having one log entry per event, a log entry resembles a batch insertion command. This is shown in figure 3.32 b). A better implementation would be to implement batching right in the log instead of outside the log. The `appendEntries` message of Raft allows to send multiple Raft entries at once. We currently don't take advantage of this. But, it is not that bad at all: many of the vendors we inspected suggest to producers to write events in batches anyway[17]. With buffering inserts, we take the burden from the event producer to send in batches by enforcing that events are written in batches anyway. While this does not reduce network latency between the client and the ChronicleDB cluster, it reduces the inter-cluster latency by a significant order.

Buffer Implementation. Figure 3.25 illustrates the buffer architecture:

(1) A client emits events to be inserted into the store. This happens either through a remote API (in this case, a `REST` API) or directly on one of the nodes via the 'Java' API in case of an embedded design.

(2) The partition manager of the ChronicleDB engine finds the right partition for the insertion request and the current leader for this replica group to serve the request. The partition and leader mapping is cached for a performanant

[17] InfluxDB, for instance, suggests as a best practice to write events in batches https://docs.influxdata.com/influxdb/v2.3/write-data/best-practices/optimize-writes/#batch-writes

in-memory lookup. The request is sent to this leader, while the client is notified for direct access of the leader for subsequent requests.

(3) The inserts are added to a buffer that allows for synchronized concurrent inserts and writes. The buffer is implemented using a `LinkedBlockingQueue`. The buffer is flushed using the `drainTo` method of the `LinkedBlockingQueue` once it overflows or at least after a certain timeout after the last insert. The buffer is optional and both its maximum size and timeout are configurable.

(4) The flushed buffer content is inserted into the event store in a batch operation. Before inserting, the events are ordered in-place in the buffer to prevent out-of-order inserts in the given set. The flush and insertion happen in a single, atomic operation.

(5) The insert of the events happens in a batch. The insert operation, including the event payloads, is appended to the Raft log, replicated and applied to the state machine. Following the Raft consensus protocol, the insertion is acknowledged once a majority of the nodes committed the operation. The state machine manages a state object, that itself encapsulates the actual ChronicleDB event store index.

Advantages of the Buffer.

- By ordering all events in a batch and ordering the batches itself (through the Raft log), we ensure linearizability.
- Our buffer implementation helps to increase the real-time consistency and to cope with out-of-order entries. The maximum size and timeout of the buffer can be configured in a way that it covers the inconsistency window, sorting out-of-order events in advance before inserting them into the actual event streams.
- The buffer also allows for concurrent writes by multiple clients, as multiple threads can write to the buffer, while its content is ordered on flush. This mitigates overlapping writes, as the buffer content is always ordered by event time, which is even more strict than the strong consistency guarantees that do not enforce ordering on concurrent writes.
- Writing events in a fire-and-forget mode to the buffer allows for very high throughput. With horizontal scaling, it is possible to keep a constantly high throughput rate (events/s). Thanks to the buffer, intermediate higher event emitting rates can be compensated, if the mean emitting rate stays below the

maximum tolerated throughput. Otherwhise, the overall system slows down and probably crashes, as the stream may not converge (cf. subsection 3.3.4.4).

Buffer Size Considerations. There are multiple factors that influence the optimal buffer size:

- **Throughput**: Since we currently rely on the basic Raft log implementation of Ratis, a sufficient buffer size should be selected to allow for high throughput. In our evaluation, we demonstrate the impact of the buffer size on the overall throughput.
- **The inconsistency window**: The buffer size and timeout can be chosen to cover the inconsistency window, so there is a high chance to provide consistency to real-time applications even with out-of-order entries.
- **The RPO**: As mentioned in subsection 2.1.6, some organizations have a metric called the Recovery Point Objective (RPO), which specifies the maximum number of lost writes that the application can tolerate when recovering from a disaster. Since the buffer entries are transient and get lost on a crash, the buffer size must be chosen accordingly. Considering some *graceful shutdown* techniques, the risk of data loss on a crash can be further reduced.

Limitations.

- As we pointed out in subsection 3.3.4.5, we should acknowledging each write to the producer by keeping a callback reference for each write. This is important when a client needs the guarantee that some events are written before it continues sending further events. Implementing this is challening and we haven't implemented that in the prototype. At present, only a successful write to the buffer is confirmed, but this does not guarantee a successful write to the quorum.
- Our buffer implementation always orders events by event time, more precisely: by the start timestamp. If a different sequence is required, it should be made configurable accordingly.
- To avoid writing big batches of events that are out-of-order when multiple clients write to the cluster, only the current leader of a Raft group should be allowed to accept these writes and add them to its buffer. For our evaluation, we ignored the multi-client case, therefore we are missing this restriction in our prototype yet. Currently, the buffer sits in the event store facade, so each node is allowed to write to its buffer and file a batch request to the leader

node once the buffers are flushed. We strongly recommend to enforce the single leader semantics.

3.3.7.8 Messaging between Raft Nodes

Apache Ratis supports a few messaging protocols for communication between nodes (for messages written to the Raft logs as well as Raft-internal messages, such as for leader election). We have chosen `gRPC`, since it comes with some benefits:

- It runs on `HTTP/2`, which has a lower latency compared to `HTTP/1.1`, and is more secure thanks to `TLS 1.2`.
- It uses `Protocol Buffers` (also referred to as `protobuf` for short) as its *interface description language*, which allows us for efficient, schema-safe messaging with out-of-the-box serialization.

All Ratis messages are defined as `protobuf` schemas, too, which makes it transparent and easy to understand what nodes exchange during Raft protocol instances.

Commands issued against state machines are represented via messages, and we define the schema of these messages with `protobuf`. Every such command must be functional, serializable and side-effect free. The messages can be sent by `RaftClients`, as shown in subsection 3.3.7.5. Example messages have been shown in the same subsection.

Messages are categorized by write and read messages. In Ratis, writes are called transactional messages, while reads are called queries, as illustrated in figure 3.29. This may be misleading since it does not provide transactions in the sense of ACID databases. It is rather to be understood as a single, atomic command execution: it is either executed and acknowledged on the state machine or no state transition happens in case of a failure. Yet this depends on the implementation of the commands and executors, and it is certainly possible to design message executors that are not atomic or deterministic, or that do not roll back changes in case of failure.

Messaging and Execution Architecture. One goal of our messaging architecture is to avoid writing monolithic state machine code. We want to decouple the replication protocol and the application logic in case we need to switch to a replication protocol other than Apache Ratis one day. Our architecture of the messages and the execution interface shown in figure 3.34 is replication-protocol agnostic. This allows us to reuse the same messaging interface for future improvements and even completely different consistency layers, e.g., the executors could be used in a job worker like

fashion to parallelize workload in eventual consistent operations, such as queries that can be served at the same time. In addition, we have schema-safety thanks to `protobuf` code generation.

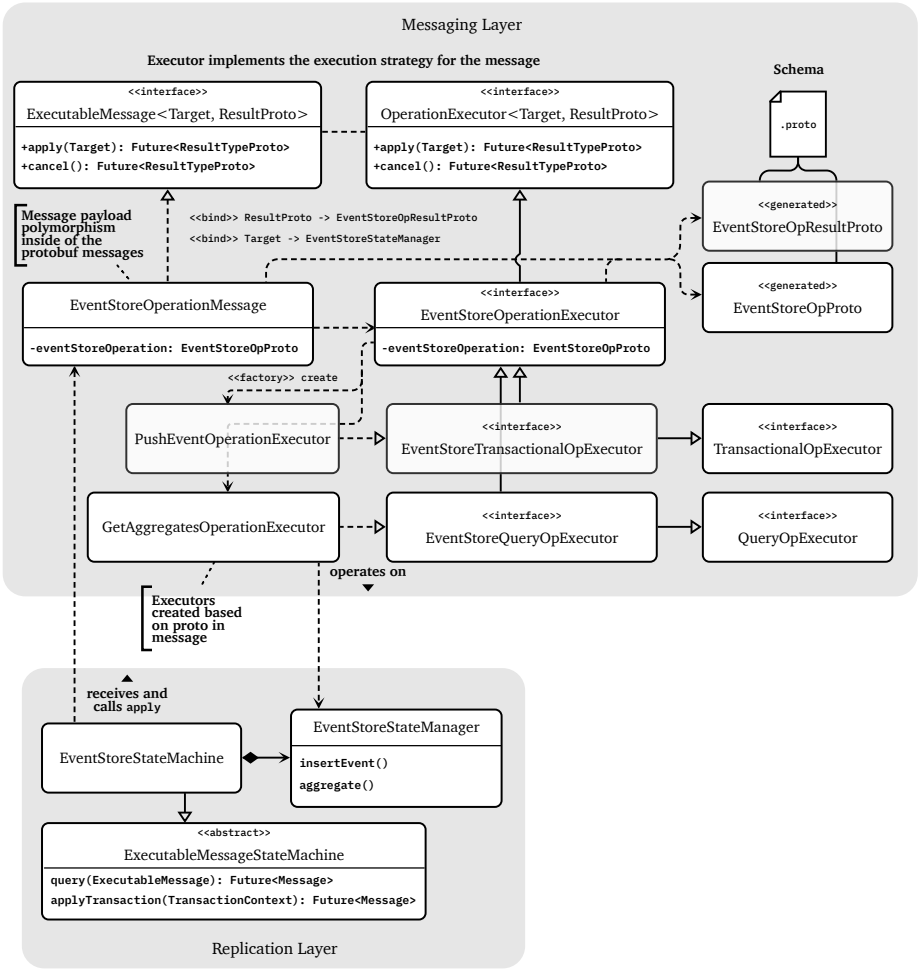


Figure 3.34: UML class diagram showing the classes and interfaces used to describe messages and operations. The message interface is implemented to wrap the schema classes generated from protobuf interface definitions, while the executor interface is implemented to describe the actual execution logic. The messages and execution logic itself is decoupled from the state machine code. This seperation allows to describe the operations agnostic from the replication protocol and implementation.

Serialization. As we already layed down, every message must be serializable so it can be sent between Raft nodes and persisted in the Raft log. It also allows us

to sent these from clients that are not written in `Java`. Since our main command is the `insert` command, we must also serialize the events that form the payload of this message. Since the payload of an event is variable, depending on the event schema, serializing it with `protobuf` is not trivial. Instead of defining a variable `protobuf` schema to serialize events, we simply use the built-in event serializer of ChronicleDB. This also prevents additional deserialization steps, since we can just pass the binary representation of the event through the message layer, until it is applied by the state machine.

Adding New Messages. Adding new messages (i.e., new supported commands for state machines) requires the following steps:

(1) Definition of the message schema in `protobuf`.

(2) Generating `Java` code (message classes) out of the schemas.

(3) Implementing the `ExecutableMessage` interface in case of an all-new state machine implementation, for example `EventStoreOperationMessage`. This wraps the generated message instances from `protobuf` and adds additional metadata and type hierarchies that are not available in `protobuf`.

(4) Expanding the factory methods of the parent message object to create the various message instances (for example, to insert an event).

(5) Implement the `OperationExecutor` interface, one for each `protobuf` message type defined. `OperationExecutors` must implement the `apply` method which defines how this single operation message is to be executed on the state machine. This method receives the payload of the `protobuf` message and a reference to the state of the state machine. There are two additional interfaces, where exactly one must be implemented: either the `TransactionOperationExecutor` or the `QueryOperationExecutor`. Only the first is allowed to mutate the state of the state machine, and it should be an atomic operation. To handle failed, non-atomic operation executions, the `TransactionOperationExecutor` interface provides a `cancel` method to implement neccessary rollbacks or compensations, in case that an operation can not be designed to be atomic and/or idempotent. It returns a `Future` containing the response of the operation for the client or an error message if it fails.

Future Improvements. In future work, we would wrap this RaftClient in a `Java` SDK as it gives us `gRPC` messaging for free and we don't need to reimplement all commands again for `REST/HTTP`. `REST` runs on `HTTP/1.1` which adds additional network round-trip and (de-)serialization time, thus slowing down the overall system. `gRPC` is more efficient than REST as it leverages `HTTP/2` and works well with our already existing binary `protobuf` definitions, without introducing additional (de-)serialization steps. However, this requires us to re-engineer the gateway implementation of the cluster manager; the clients must first ask the cluster manager which node they should talk to before sending the request. We sketched this in figure 3.35. Currently in our demo application, this is handled on the server side by the `ChronicleEngine` for simplicity reasons. This also allows us to avoid another additional serialization step when deploying ChronicleDB as a service, since clients could directly serialize into ChronicleDB's target format instead of a `JSON` string first, which must be deserialized and serialized again on server side.

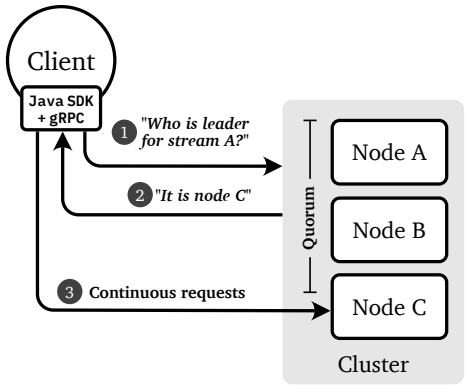


Figure 3.35: The bottleneck of json serialization of events and additional network round-trip can be reduced by introducing a gRPC API. This requires a discovery step where the client SDK asks the cluster for the leader and Raft group ID of the stream it wants to talk to.

There is great potential for the application of model-based programming and code generation. We have shown how to add additional messages, which can quickly become cumbersome. With the `protobuf` definitions and additional `Java` annotations, one could imagine to generate a huge portion of this code—including messages, message wrappers, factories, and operation executors.

We also recommend readers to keep an eye on the further development of `gRPC` regarding support for the new `HTTP/3` protocol[18], which is built on top of `QUIC` and

[18]The request for `HTTP3` support in `gRPC` is tracked in this issue on GitHub: https://github.com/grpc/grpc/issues/19126

UDP instead of TCP. QUIC was approved as an IETF standard in 2021 [191], while HTTP/3 was approved recently in June 2022 [192]. HTTP/3 can further reduce latency, especially in settings with a lot of roundtrips, such as in Raft.

3.3.7.9 Partitioning and Horizontal Scalability

The throughput of a single event stream instance has an upper bound, based on I/O capabilities of the underlying machine, the OS, concurrent access to the ressources of that machine, the implementation of the ChronicleDB index and lastly—usually with the most impact—the replication protocol and the underlying network. To be able to scale beyond this limit and to overcome the latency trade-offs introduced by Raft, we need to introduce partitioning (cf. section 2.1.5).

With partitioning, we can make the distributed ChronicleDB horizontally scalable. For our prototype, we implemented a basic approach that runs one partition per event stream, with simple load balancing based on the number of available nodes and required replicas.

Multi-Raft. Ratis allows for Multi-Raft, which supports running multiple Raft groups on a single clusters. When you run multiple basic Raft instances, one for each partition (stream), on a single cluster, each of these comes with their own leaders and heartbeats. Unless we constrain the Raft group participants or the number of topics, this creates an explosion of network traffic between nodes. In Multi-Raft, this is handled differently. There are dedicated leaders for each group, but only a single heartbeat round-trip for all nodes. A node, once it partakes in at least one Raft group, becomes a Raft server. The Raft server is responsible for sending and/or receiving AppendEntries messages and hearbeats (empty such messages). The Raft server itself is logically split into multiple Raft divisions. One division represents one Raft server taking part in one Raft cluster. On the division level, each server can be either a leader, follower or candidate. We illustrate our basic Multi-Raft approach in figure 3.36.

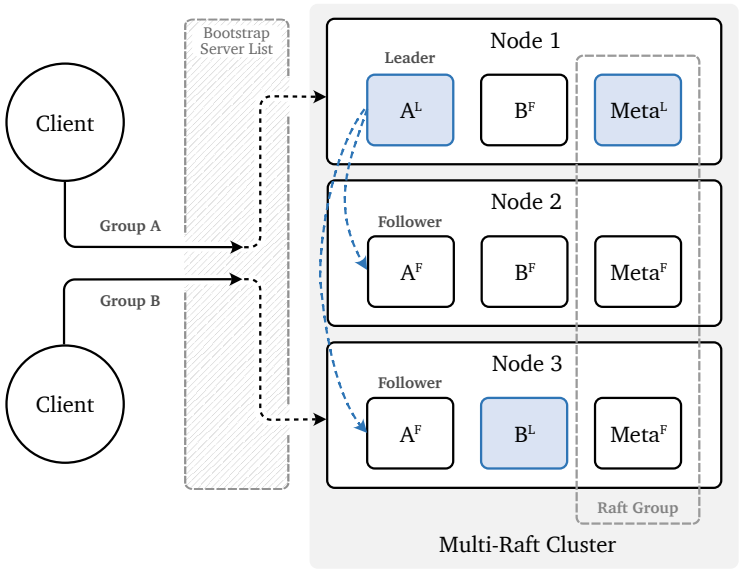


Figure 3.36: Partitioning and replication with multiple Raft groups. When a client accesses a a Raft group on the cluster for the first time, it accesses any available node it has on its bootstrap list. If this node is not the leader, it updates the clients bootstrap list. Subsequently, the client writes and queries this leader node until a new leader has been voted. A node can participate in multiple Raft groups, having either a leader, follower or candidate role for this group.

Load-Balancing. With Multi-Raft, we can achieve very basic load-balancing by evenly distributing replicas on available nodes. We achieve this using a `PriorityBlockingQueue` that orders nodes by the number of already registered groups on that nodes. We adopted this approach from the Apache Ratis examples. This is illustrated in figure 3.37, where 6 partitions (including metadata/cluster management) are distributed on 5 nodes with a replication factor of 3 (note that the cluster management is deployed on each node to increase its availability, fault-tolerance and locality).

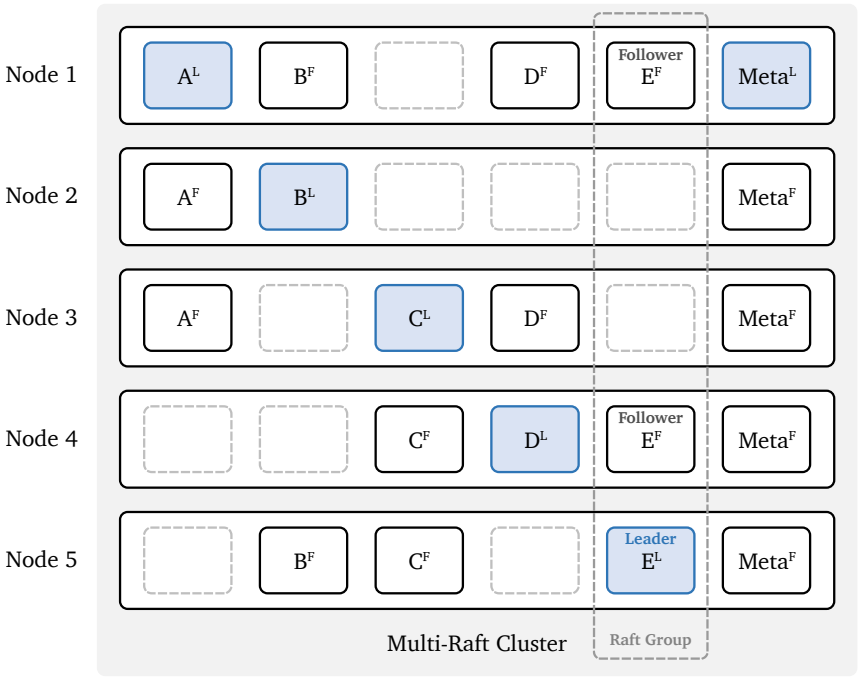


Figure 3.37: Basic load-balancing with Multi-Raft. Partitions with a replication factor of 3 are evenly distributed across 5 nodes. Metadata is replicated on every node to ensure cross-cluster disaster resilience and fault tolerance and allows the partitions on the respective nodes to directly read the metadata on its own node.

In the next paragraphs, we take a look at further possible improvements, that did not make it into our prototype.

Time Splits. Since event streams are append-only structures, historical data quickly becomes immutable (at least after the inconsistency window). In addition, ChronicleDB splits the stream storage by time splits (see section 2.4.4.1 for reference). We can take further advantage of this by introducing partitioning by those time splits. The time splits in ChronicleDB sit in their own TAB$^+$ index trees, as well as in seperate files, which makes it naturally suitable for partitioning.

Time splits allow for improved read scaling. Historic time splits are read-only (we will assume that out-of-order events do not happen in historic splits), thus the single-leader requirement of Raft can be dropped here. Furthermore, historic time splits could also be replicated using a different replication protocol, such as primary-copy, or simply by installing Raft snapshots, since we don't care about write consistency anymore. Generally, they would naturally "grow", once the index tree

is “chopped” when a new time split should be created. When this happens, a new partition could be instantiated immediately (or better in advance to stay available during this period) and the cluster manager routes writes to the new partition, while the old partition is locked for writes and made available to be read on all nodes. We sketch such a configuration in figure 3.38. Moreover, the replication factor can also be increased for historical splits by installing snapshots, which further reduces read latency and provides data locality in the case of multiple availability regions.

We haven’t implemented partitioning by time splits yet, since we need some strategies to execute queries and aggregate requests across time splits.

Partitioning by time splits is also available in other time series databases and event stores, such as IoTDB (there, it is called *time slices*) [158].

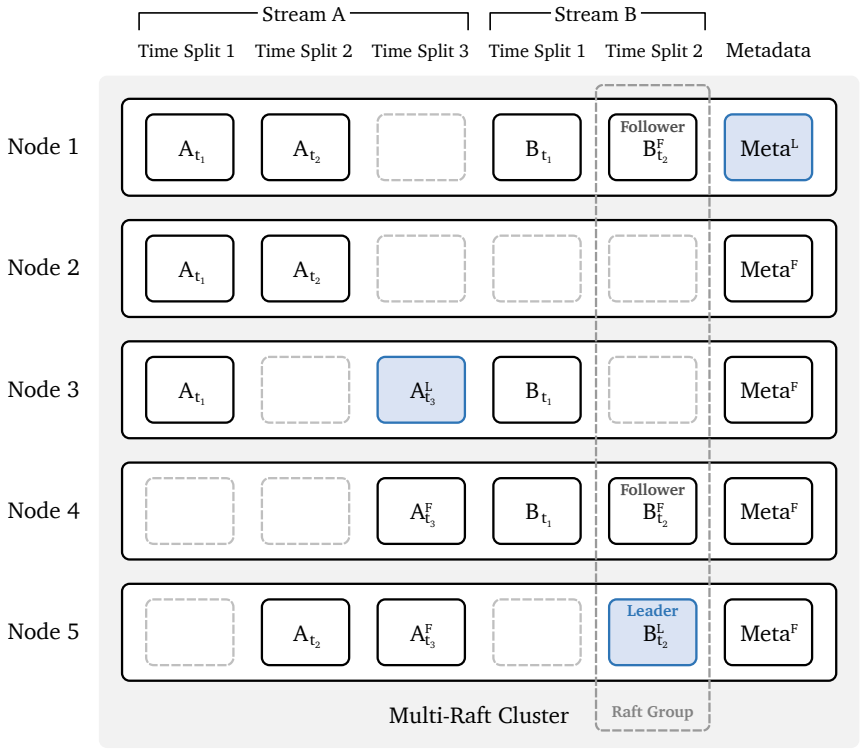


Figure 3.38: Partitioning using time splits with a replication factor of 3, load balanced on 5 nodes. For historic time splits, the single-leader requirement could be dropped since they are read-only, resulting in that every replica can serve query requests. The superscript here describes if a node is either a leader or follower, while the subscript denotes the starting timestamp of the split.

Sharding (Write Splits). While partitioning by time splits lowers the read latency, sharding (effectively resulting in *write splits*) lowers the write latency, allowing the event store to scale with the number of writing clients. We illustrate a possible setup for sharding in figure 3.39.

Sharding in consistently ordered append-only structures is not trivial. Since sharding would only lower write latency when writes across shards are no longer linearized, write consistency must be sacrificed. Instead of strong consistency, each partition would only offer sequential consistency at maximum. Therefore, we haven't implemented this in our prototype. While sacrificing linearization on writes, we need to re-introduce linearization on reads, at least for the recent time split.

- When historic time splits are generated, the shards of the current split could be merged in the background into a buffer (i.e., a ring buffer) while another thread writes the immediate results into the historic split.
- To achieve linearizability on recent splits when querying shards, we need to merge shards of the same event schema in the query engine. Thanks to the sequential ordering of each shard, we can just use in-memory merging and cache our results for already merged intervals (the cache must be partially invalidated if out-of-order events occur). For instance, merging using a min-heap can be done with $\mathcal{O}(Nk \log k)$ time and $\mathcal{O}(Nk)$ space complexity, where N is the number of events per shard (in case of balanced shards) and k the number of shards. With tumbling time windows and parallelization, we can greatly improve the time complexity, since individual time windows be merged independently. We illustrate this in figure 3.40.

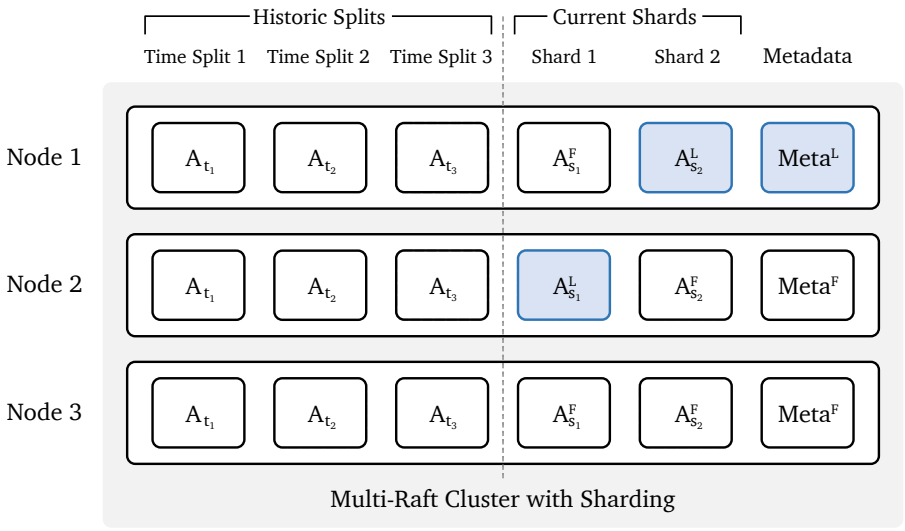


Figure 3.39: A multi-raft cluster with historic time splits and sharding of the current time split to improve write throughput and latency

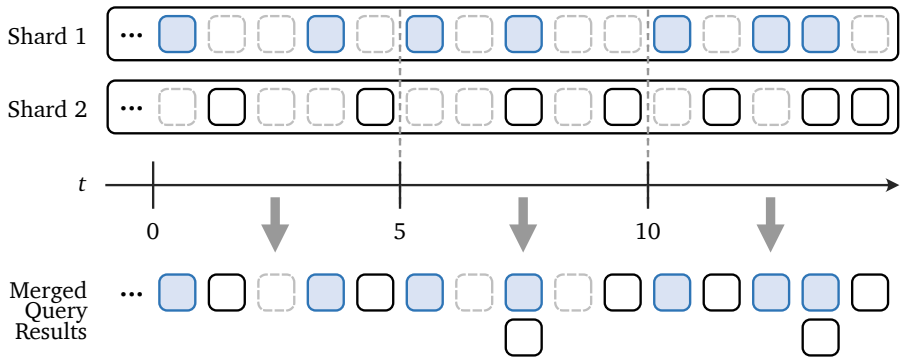


Figure 3.40: Illustration of two event stream shards merged on a query with tumbling windows

Consistency Across Partitions. With our partitioning approach, we do not provide strong consistency across streams, since we do not ensure that the order of events to be written into each event stream matches their insertion time (cf. subsection 3.3.4.6). This is fine, since we are only interested in the actual event time. If there is causality between streams, this only becomes relevant when querying across multiple such streams and should be handled by the distributed query processor, as shown in figure 3.20 (also see the previous paragraphs for a parallelizable merge approach). We haven't implemented this in our prototype in the context of this work.

For the sharding case, it looks as follows: even though sharding (in combination with horizontal scalability) significantly increases overall throughput, it causes a stream to lose write consistency as we forgo inter-shard coordination—which would only increase latency again. We need to restore the consistency again when querying the shards, i.e. by merging. How this could look like architecture-wise is illustrated in figure 3.41. We also described this in the sense of causal consistency in subsection 3.3.4.2.

Even across streams for different event schemas, consistency cannot be guaranteed because eeach stream is handled by its own partition and thus its own Raft group. We limit our implementation to the one-partition-per-stream case, thus we are not able to evaluate the impact of sharding on the overall througput.

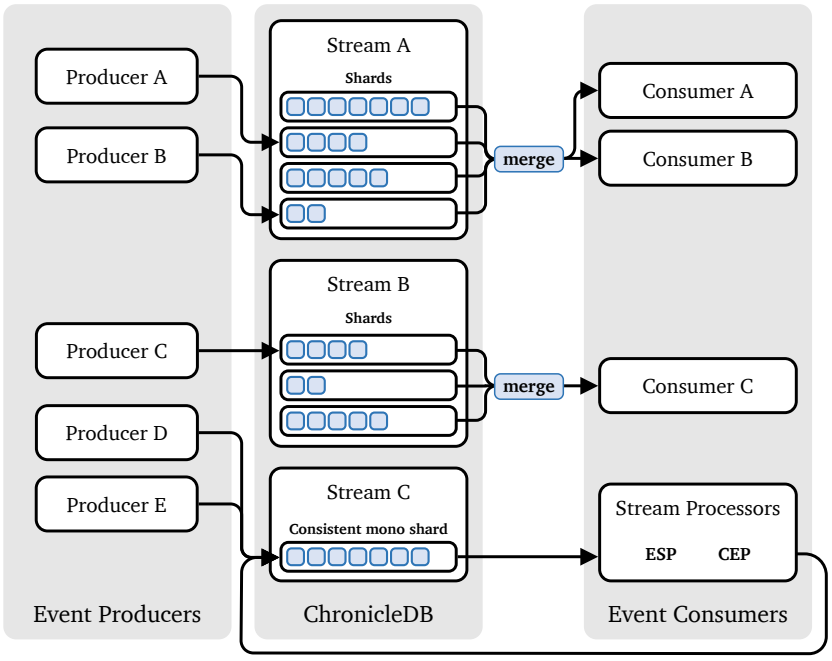


Figure 3.41: Schematic illustration of the relation between producers and consumers, and how consistency across shards may look like. The merge step is also illustrated in figure 3.40.

Rebalancing. Once a node crashes, it should be replaced by another available node, to ensure a quorum can still be maintained for each Raft group. There are two cases of node crashes from the perspective of a single Raft group: The crash of a follower and the crash of the leader. On both, the network reconfiguration protocol of Raft must come into play to allow a new node to join the group, while

the protocol differs on both cases. The network reconfiguration in Raft allows for rebalancing without compromising availability. We illustrate these cases in figures 3.42} and 3.43. Note that our prototype is missing the rebalancing yet.

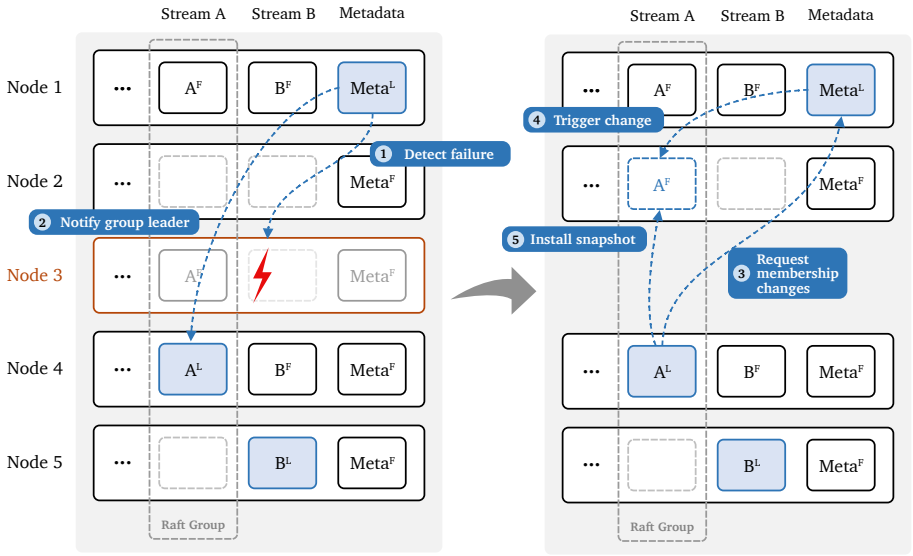


Figure 3.42: Rebalancing in the event of a node failure that leads to a raft group with too few replicas, while the leader stays intact. a) A fault makes a node crash (fail-stop). The meta quorum detects this crash and notifies the leader of this group. b) The leader requests a membership change from the meta quorum. The quorum selects a node based on balancing rules and triggers a network reconfiguration. The node is added as a new follower to the group. The leader then installs the current state snapshot on this new follower replica.

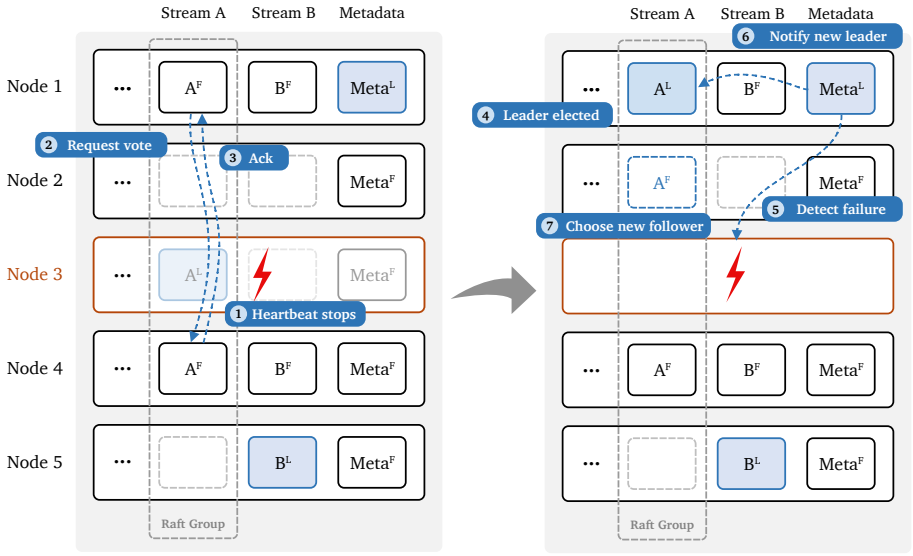


Figure 3.43: Rebalancing in the event of a leader failure. a) As the followers no longer receive heartbeats from the leader of their group, they start a vote. The winner of the vote becomes the new leader of this group. b) The cluster manager can now assign a new follower similar to the case in figure 3.42.

3.3.7.10 Edge-Cloud Design

By adding a replication and partioning layer to ChronicleDB, while keeping its capabilities of running standalone on a single server or embedded in another application, ChronicleDB meets the basic requirements to run in an edge-cloud architecture.

We illustrated the edge architecture, including its three layers, in section 2.1.7. ChronicleDB fits into all three layers, thanks to its different deployment modes. An example setup for the edge-cloud is illustrated in figure 3.44.

Terminal Layer. ChronicleDB is originally designed and optimized to run on the terminal layer. It can be used as a "centralized storage system for cheap disks running as an embedded storage solution within a system (e.g., a self-driving car)" [10]. It then runs as a lightweight library tightly integrated in the application code, with zero network latency, allowing for the highest throughput on the perspective of a single stream.

Edge Layer. On the edge layer, ChronicleDB allows to aggregate a multitude of events from devices in the terminal layer. In general, the complexity of the events

between the layers increases, while the volume per stream decreases: While the terminal layer produces raw events, the edge layer provides filtered and aggegrated events to the cloud layer that are ready to be consumed in reporting applications. The basic idea is to run *Event Processing Agents* (EPA) in the sense of Complex Event Processing on the edge layer and use ChronicleDB in standalone mode as an event sink. By running ChronicleDB on the edge, it provides highest data locality and event processing power close to clients, providing lower response times since dedicated ressources on the edge can be used for processing that are not available in the cloud to these extents.

Cloud Layer. On the cloud layer, ChronicleDB is deployed in the distributed version that we presented in this work. In the cloud, the goal is to scale with the number of devices, respectively streams, from the terminal layer, while providing high-availability and fault-tolerance. While ChronicleDB in the cloud does not provide the same high throughput on a per-stream basis as embedded on a device in the terminal layer, it shows its advantages in horizontal scaling, as it beats the throughput of the embedded version on a multi-stream basis.

Between the Layers. To close the gap between the ChronicleDB instances on the different layers, one could either use an event processing solution, a pub/sub system or continously querying one ChronicleDB deployment with another. The latter is only suitable between the edge and the cloud, and to stream edge or cloud events into the terminal layer, since we can not query streams from devices on the terminal layer in a pull fashion—this is better to be implemented in a push fashion. We refer at this point to the related literature.

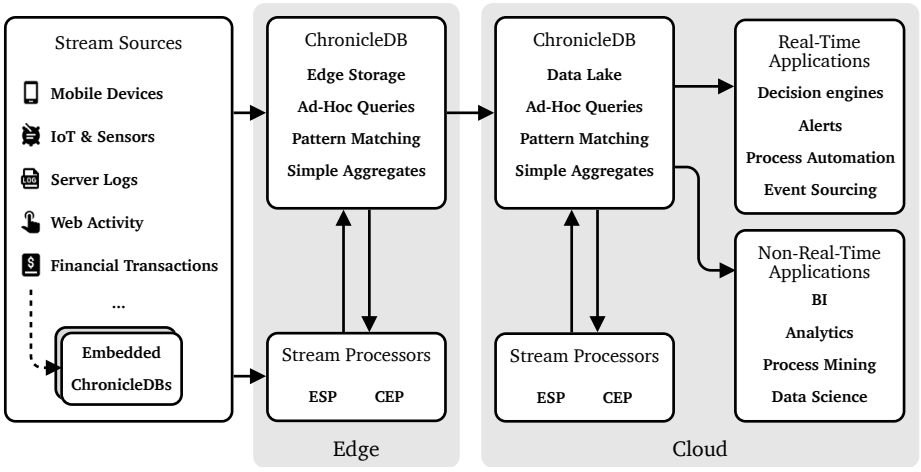


Figure 3.44: Potential architecture of running ChronicleDB in an edge-cloud setting, next to and in combination with stream processing systems. It can act both to ingest and serve data before processing, as well as a data lake after processing. Different applications can query the ChronicleDB event store, either ad-hoc or continously. Both real-time and non-real-time systems can read streams from ChronicleDB, which can result in different consistency levels required.

3.3.8 Evaluation Application

To test and evaluate the implementation of the replicated event store and the middlewares supporting it, a test application is needed. In the context of this work, a synthetic application is implemented to perform trade-off studies. The application is build with the following stack:

- Spring Boot as the `Java` Framework to provide ChronicleDB on a Raft as a service.
- React to provide a lightweight user interface frontend for evaluation purposes.
- Maven and Webpack to build the whole application.
- Docker to run it in containers.

This setup allows us to evaluate ChronicleDB both locally and in the cloud.

3.3.8.1 Supported ChronicleDB Feature Scope

We do not support all baseline features of ChronicleDB now in our distributed prototype. We support the following two main operations that we used to evaluate our implementation:

- **Insertions**: We support inserting events, either as single events, buffered, or in custom defined batches.
- **Aggregation**: We support basic ad-hoc aggregation by implementing the `AggregatedEventStore` interface and using the aggregate feature of the `TABPlusTree` under the hood. We use this to evaluate the read performance.

Limitations. We do not support queries at the moment, neither ad-hoc nor continuous queries. See section 3.3.7.1 for details.

3.3.8.2 Running Inserts

The evaluation application does not use the `HTTP REST` API to insert events, as `REST` runs on `HTTP/1.1` which adds additional network round-trip and (de-)serialization time, thus slowing down the overall system (cf. subsection 3.3.7.8). Instead, we run an evaluation service on one node of the cluster, that continuously generates random events and inserts them into the replicated event store to simulate a continuously emitting sensor device. This evaluation service can be started and stopped via a dedicated API, given params to control the velocity of inserts. In addition, we have written some simple `python` functions to call this evaluation service from a `Jupyter` notebook, not matter if ChronicleDB runs on a local or remote cluster. The notebook allows us and others to run the evaluation reliably and repeatable. This notebook can be found in the appendix.

3.3.8.3 Running a Cluster

Deploying and running a ChronicleDB cluster looks similar to how you run any other distributed database cluster. When starting a node, you specify its ID in the cluster, its storage directories, the ports for metadata/management, replication and the public API, and tell each node the adresses of other nodes in the cluster (the bootstrap list of at least 3 nodes to form a quorum) so they can find each other (while this list is further populated by the cluster manager once a quorum of nodes can talk to each other).

For instance, a single node on a local cluster (single machine) is started this way:

```
PEERS=n1:localhost:6000,n2:localhost:6001,n3:localhost:6002

ID=n1
SERVER_PORT=8080
META_PORT=6000
```

```
java -jar target/chronicledb-raft-0.0.1-alpha.jar \
--node-id=$ID \
--server.address=localhost \
--server.port=$SERVER_PORT \
--metadata-port=$META_PORT \
--storage=$STORAGE_DIR/$ID \
--peers=$PEERS
```

This makes it easy to run a ChronicleDB cluster in a container environment, such as docker. A whole cluster can be deployed with a single docker-compose.

3.3.8.4 User Interface

The prototype comes with a simple user interface that helps to access various functions the cluster manager, such as checking the vitality of the cluster. It serves an overview over all running raft groups and the current roles of each node in that groups. It also allows to manage event streams and to run aggregations on them, mainly to continuously observe the throughput in evaluation settings. It is also used to evaluate and demonstrate the aggregations themselves.

It allows navigation between cluster nodes and can also be used to check the impact of the cluster's strong consistency from the user perspective, simply by running the user interfaces of all nodes in multiple browser windows and observing how changes are rendered simultaneously.

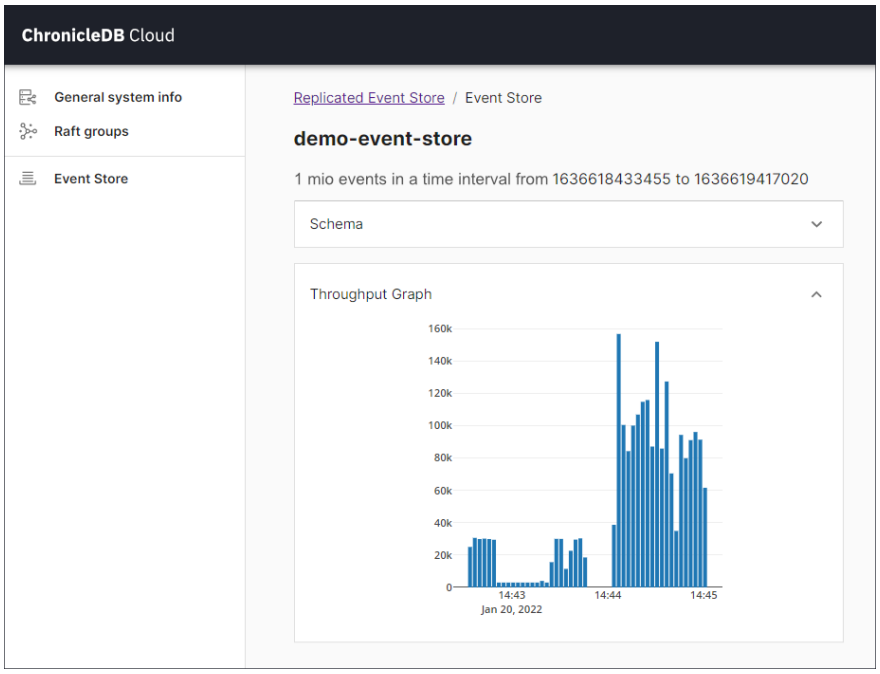


Figure 3.45: Screenshot of the ChronicleDB event store UI

3.3.9 Implementation Summary

In this subchapter, we presented our implementation of a replication and partitioning layer for ChronicleDB. We went over the system design, starting with a thorough discussion of consistency models and their impact on ChronicleDB as well as on clients. We worked out the requirements for the replication layer and decided on a consistency model that would meet those requirements. We decided for strong consistency, since this allows us to guarantee safety to clients, at least in the absence of out-of-order events. We discussed the impact of out-of-order events on write and read consistency, as well as various mitigation strategies, and decided to implement a write buffer as a simple and lightweight solution. We laid out the trade-offs of this solution and made suggestions for possible future improvements. Finally, we explained our choice for a replication protocol and a library that implements this protocol in `Java`. Based on this library, we presented the technical architecture of our prototype implementation, which also includes an evaluation service.

In the next subchapter, we will use this evaluation service to evaluate our implementation, using the original standalone ChronicleDB implementation as a benchmark.

Chapter 4

Evaluation

I've always liked code that runs fast.
— Jeff Dean

In this work, an implementation of the Raft consensus protocol was proposed for replication of ChronicleDB, a high-throughput event store. Since Raft is strongly consistent, a trade-off in performance is to be expected (see section 2.2.11). In theory, the larger the number of replicas, the higher the data availability; however, at the same time, the cost of replication (which comes at the expense of performance) increases. In this section, we present the results from our evaluation of the prototype implementation, specifically the cost of replication, by quantifying the costs and benefits that come with this approach. The challenge of the implementation is to find an optimal trade-off between the cost of replication and the availability and consistency of data.

4.1 Evaluation Goals

The main goal of this work is to test the additional cost of our techniques in terms of increased latency and reduced throughput in the system. We want to identify potential bottlenecks and ways to improve the prototype without sacrificing consistency. We evaluate different cluster settings and compare them with the standalone ChronicleDB to describe the impact on performance.

4.2 General Performance Considerations

Before running our evaluation, we set some expectations based on roundtrip and I/O considerations of Multi-Raft. Raft introduces at least one round-trip to a quorum of the nodes with the least delay when replicating a single log entry, and introduces additional I/O for writing log entries to disk—and probably less efficient in Ratis then in ChronicleDB itself. From what we've learned from other distributed event-store-alike systems, the throughput with strong consistency for a single, unpartitioned stream ranges from $\sim$ 330,000 operations/sec for InfluxDB, partially running on Raft + Primary-Copy, to $\sim$ 1,000,000 operations/s for TigerBeetle on Viewstamped Replication (while this actually accounted for transactions, not just single events), running on comparable setups (cf. subchapter 3.2). We also expect a linear correlation between the number of nodes in a cluster and decreased write throughput, based on our observations.

4.3 Limitations

Read Saturated Systems. In our experimental setup, we only measured write performance in isolation. We did not measured write performance in read-saturated systems—which comes closer to a real-world usage example.

Recent Improvements. We already identified bottlenecks in our implementation (cluster management and application logic servers too tightly coupled, redundant serialization, and an inefficient Raft log implementation) and started working on them. To the time of this writing, we didn't finished our work, though. We would expect these changes to improve the throughput by a significant number, but we can not validate this yet.

Multi-Partition Performance. With multiple partitions, the write throughput is expected to increase because write load can be distributed across leaders. Unfortunately, at the time of this writing, we did not managed to finish our work on instantiating multiple partitions and streams in our prototype, and therefore, we can not validate this.

Upscaling Cluster Machines. We evaluated throughput only on very cheap machines on AWS. We did not measure the impact of upscaling these machines. The effect of upscaling should be evaluated to compare the benefit in throughput with

the additional cost of hardware (multiplicated with every replica).

4.4 Setup for Comparison

We run our evaluation application (see section 3.3.8) on two different machine setups. The first setup is a deployment on a local machine, and the second setup is a deployment on seperate machines in the same availability zone.

Table 4.1: Setup for evaluation on a local machine

Code	ChronicleDB Cloud v0.0.1-alpha[1]
OS	Microsoft Windows 10 Professional 64 Bit
CPU	Intel Core i7-10510U (4 Cores)
Disk	SSD (PCIe NVMe)
Memory	32 GB DDR4 SDRAM
Network	Protocol Buffers over Localhost Loopback

Table 4.2: Setup for evaluation on a remote cluster

Code	ChronicleDB Cloud v0.0.1-alpha[2]
OS	Amazon Linux 4.14.262-200.489.amzn2.x86_64
CPU	1 vCPU
Disk	SSD
Memory	2 GB
Network	Protocol Buffers over HTTP/2, $\sim$ 500 MB/s up-/download
Region	eu-central-1

We also run these two cluster setups with different replication factors and buffer sizes.

Simulation Data. We simulate stock price data by creating random events of the following schema in advance, holding them in memory on one node and then inserting them into the `ReplicatedEventStore`.

```
SYMBOL: String,
```

```
SECURITYTYPE: Integer,
LASTTRADEPRICE: Float
```

We can run one trial of such an experiment by first creating a stream partition for the target schema and then calling the REST endpoint that starts the evaluation run. This evaluation service measures and returns us the total runtime of inserting the events. The whole experimental setup and evaluation can be found and repeated by yourself in the Jupyter notebook in the appendix.

We run all our trials on a single stream only. We expect higher throughputs when writing to multiple streams, since this would leverage the write load balancing of Multi-Raft.

4.5 Dependability

Since we use Apache Ratis on the replication layer, most of the dependability results are based on the Ratis implementation. Unfortunately, Apache Ratis does not come with a validation of liveness and safety, so we assume that it follows the Raft TLA$^+$ spec, but we also expect bugs to appear that may not be easy to detect. Therefore, our conclusion on each dependability property is:

- **Availability**: Distributed ChronicleDB is available as long as a quorum of nodes is available.
- **Reliability**: As long as the maximum throughput has not been exceeded for an extended period of time, the cluster operated reliably. With throughput spikes for a longer period of time, the cluster crashed and had to be restarted (see section 4.8).
- **Safety**: Raft provides strong consistency, so as long as Ratis follows the full specification, all nodes will follow the same operation order, therefore exposing equal event stores. We did not implemented a byzantine-fault tolerant Raft, so tampering is still a possible thread. Therefore, we do not recommend to allow distibuted ChronicleDB to run without permissions.
- **Integrity**: As long as Ratis follows the full Raft specification, integrity should not be violated.
- **Maintainability**: Our modular architecture generally allows to maintain a state machine instance without disrupting other running instances.

4.6 Ratis Performance

Running our prototype without a buffer has shown that Apache Ratis introduces a huge decrease in throughput. On a local machine with three virtual nodes (see table 4.1), our implementation of Ratis using the naive default log implementation yields a maximum throughput of 250 operations/s with a median of 95 operations/s. As the Ratis team claims, the naive log should not be used for write-intense applications, and custom implementation is mandatory to serve high throughputs (see section 3.3.6.1).

4.7 Throughput on Different Cluster Settings

To evaluate our prototype, we measure the throughput when the system is saturated with writes.

4.7.1 Comparison of Cluster Sizes

We compare different cluster sizes on a local machine (see table 4.1). So far, this result must be taken with a grain of salt, since we did this comparison on a local machine only, which quickly hits I/O boundaries of the machine.

In figure 4.1, we plot our results of the performance comparison of clusters with different replication factors.

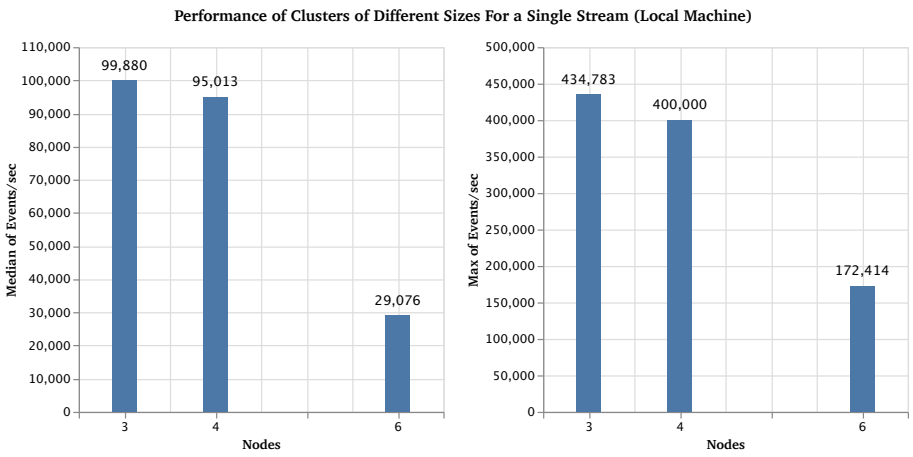


Figure 4.1: Comparison of the median and maximum performance of different cluster sizes for a single stream on a local machine.

We expected a linear correlation between the number of nodes in a cluster

and decreased write throughput, similar to other distributed storage systems. We actually observe a slightly logarithmic trend, which we assume is the case because we approached maximum I/O on our single machine. Since we did not evaluate this on clusters with actual different physical machines, we can not validate this hypothesis here.

4.7.2 Comparison of Buffer Sizes

Without a buffer, writing every event to the Raft log slows down the system tremendously, due to increased I/O and especially because the default log in Ratis requires a log entry to be written before it is sent to other nodes, effectively blocking the whole system on every single operation, as we already mentioned.

We compared different buffer sizes of 1KB, 10 KB, 100 KB and 1 MB. Unsurprisingly, a larger buffer yields better performance, but from our results in figure 4.2 it seems that it approaches a certain limit, that is far from the performance that we achieve with standalone ChronicleDB. We show the different buffer sizes on a log scale, and the median and max throughput on a linear scale.

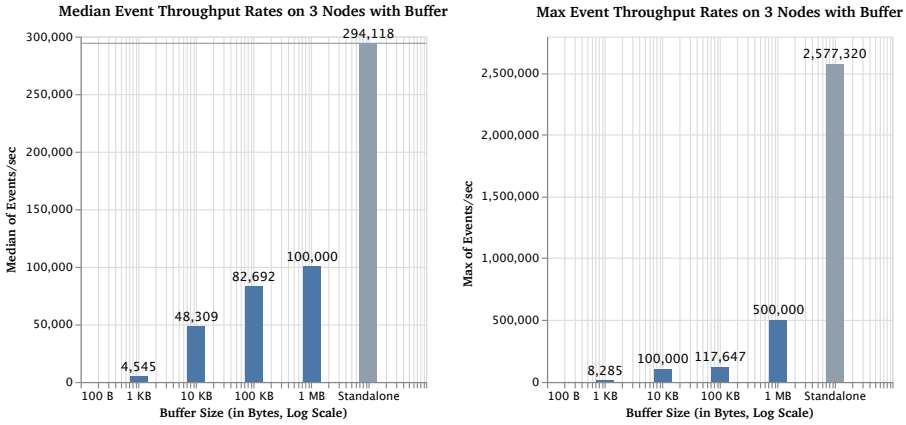


Figure 4.2: Comparison of the throughput of different buffer sizes with standalone ChronicleDB as a benchmark

To put our results in a different perspective, we show in figure 4.3 the median time to ingest 1,000,000 events.

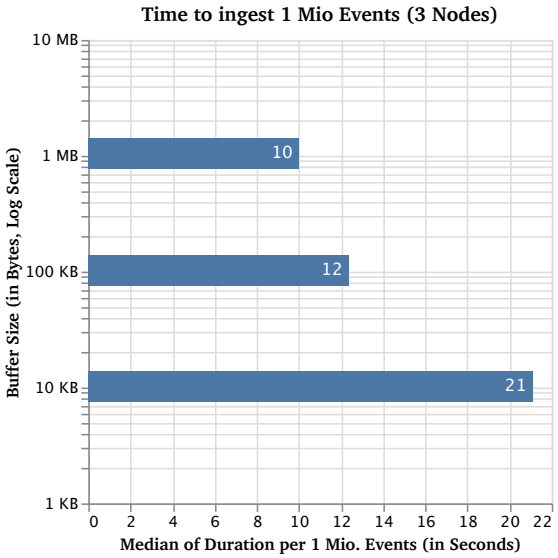


Figure 4.3: Median time to ingest 1 mio events for different buffer sizes

The boxplots in figure 4.4 in addition show how our distributed ChronicleDB prototype competes against a standalone deployment in both median and maximum throughput. While the median throughputs are relatively close, the maximum throughputs differ significantly. We explain this with our observations in section 4.8.

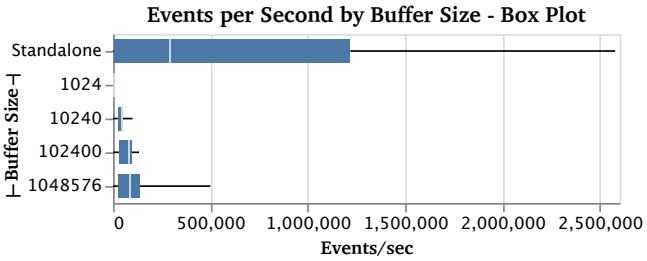


Figure 4.4: Comparison of the throughput of different buffer sizes with standalone ChronicleDB as box plots

The size of a log entry in Ratis is limited. We tried to increase it, but somehow failed. So, the maximum size of a log entry is what limits our maximum buffer size, since we do not extract single events into single log entries and send them in batches yet (see section 3.3.7.7).

4.7.3 Throughput on a Distributed Remote Cluster

We evaluated also on a remote setup (see table 4.2), which yields slighlty better results compared to those on our local setup. Since we used very cheap machines on AWS to run this evaluation, compared to our local machine, we interpret these results as a success; however, we can not validate if upscaling single machines in the remote cluster will improve the performance since we did not measure that. If upscaling would not improve throughput, the bottleneck is clearly in the replication layer.

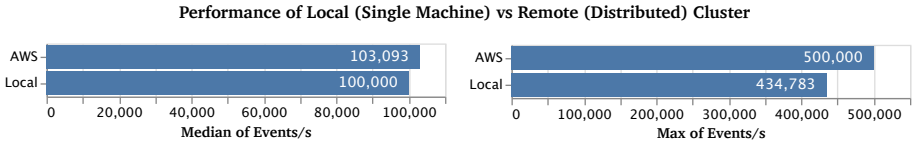


Figure 4.5: Performance of local vs. remote cluster

4.8 Exceeding the Upper Throughput Threshold

We stress-tested the cluster by increasing the write load in a way that the mean throughput exceeds the maximum throughput for a longer period of time.

The stress-test rendered our cluster completely unavailable. It revealed that with spikes in throughput that are above the maximum tolerated throughput, the chance of blocking all of the the leader's threads increases. When this happens, the leader is less likely to send heartbeats in time. This triggers the leader election protocol. Leader election adds even more latency, while write requests are still ongoing, and causes further client requests to accumulate, ending up in a loop of leader elections and finally a complete crash of the system.

While the buffer mitigates this to a certain degree, it is also only able to cover a few spikes in throughput, but can not prevent this behavior in long-term throughput overruns.

4.9 Benchmarking against Standalone ChronicleDB

As shown in figures 4.2 and 4.4, The performance of standalone ChronicleDB for a single stream is superior to that of the distributed solution. With a replication factor of 3 and a buffer size of 1 MB, the median throughput is up to 3 times higher on standalone ChronicleDB, and the maximum throughput is more than 5 times higher. We assume that this can be outweighted quikcly as soon as multiple streams

are served, because distributed ChronicleDB can balance the write load on multiple leader nodes. We even do not consider this to be bad, since we haven't implemented any optimizations yet, such as a pointer-only Raft log with concurrent access, and this results still work in a edge-cloud setting, where sensors run the embedded, high-throughput version of ChronicleDB, and only aggregated events will be written to the distributed deployment(s).

4.9.1 Querying and Aggregation

We implemented the aggregate interface in our distributed ChronicleDB prototype. We continuously ran aggregates with our evaluation application. We did not see any significant differences in read performance compared to standalone ChronicleDB.

We have not implemented the query interface (see section 3.3.2), so we can not reason about this here.

4.10 Profiling

We profiled our implementation to find the most CPU and I/O intense steps. We found that event serialization takes up a good amount of time, since we serialize events of a batch write sequentially on a single thread. We also use parts of the event serializer to estimate the size of an event before writing it into the buffer. We could omit this step by allowing the buffer size to be configured by maximum event count instead of bytes, but since events can have different schemas and variable sizes (when strings are used), we need to estimate the buffer size to not exceed the maximum size of a Ratis Raft log entry.

4.11 Comparison with other Distributed Event Stores and Time Series Databases

Compared to other distributed event stores and similar systems, we achieve similar results on the lower end of the range (cf. section 4.2). Since we did not implemented any further optimizations yet, we assume that distributed ChronicleDB is actually capable of outperforming this systems. We can learn from the implementation of the replication layers of those competing systems, and standalone ChronicleDB outperforms most of other standalone systems, so this should be possible.

Chapter 5

Conclusion

> All software architectures capture tradeoffs, and deciding between alternatives is about choosing which tradeoffs you can live with in a particular context.
>
> — Jimmy Lin

This work covered the topic of replication for distributed event stores and comparable data stream management systems. We discussed the terms dependability, fault-tolerance and consistency and compared various replication protocols. Based on the potential use cases of a event stores and event processing systems that could be used in edge-to-cloud networks, we found that strong consistency satisfies all cases, even if it increases the total cost of replication, namely the write latency.

With strong consistency, we arrive somehow at a one-size-fits-all solution. It works in every case, but not perfectly. The trade-offs in latency and availability may make this approach not suitable enough for extremely high-throughput cases without further optimizations, but there are multiple other strategies to mitigate this, such as partitioning, which we introduced in our implementation. It may sound pessimisticly, but you can't have it all: the CAP and PACELC theorem have proven that in theory, and many implementations have shown that in practice. But actually, it is not that bad at all: even if there will always be a decision to be made between generalized vs. specialized implementations, both offer better dependability and scalability properties than a single-node deployment may ever be possible to serve.

As a result of our research, we presented ChronicleDB on a Raft, a Raft-based replication layer for the high-performance ChronicleDB event store. We have shown that we can provide a replication layer that allows us to deploy ChronicleDB on a distributed cluster, while providing strong consistency to clients—to some degree

even for out-of-order events. We evaluated ChronicleDB in regards of the expected trade-offs. We have run all our evaluations on a single stream only. Both the median and maximum throughput for a single stream is significantly lower (almost 3 times for median and 5 times for maximum throughput) compared to standalone ChronicleDB on the same machine. We expect higher throughputs when writing to multiple streams, since this would leverage the write load distribution of Multi-Raft. With such horizontal scaling, we would expect to outperform standalone ChronicleDB. Since our prototype implementation is far from being optimized, we expect that a production-ready distributed ChronicleDB yields better performance, even on a per-stream level. In addition, running ChronicleDB on a distributed cluster increases hardware and infrastructure costs, but this is negligible compared to the benefits in achieving fault tolerance and availability.

5.1 Recommendations and Future Work

Strive for progress, not perfection.

We have noticed that our solution leads to a significant performance drop. Since we did not implement any optimizations other than a simple buffer, this is in line with our expectations. We have discovered possible optimizations and recommend system developers to take them into account when implementing a production-ready distributed event buffer.

5.1.1 Consistency Considerations

Some related data stores such as InfluxDB show that strongly consistent replication of all data with a consensus protocol can be impractical and slows down the system, at least if no further optimizations are made in the replication layer. Others, such as TigerBeetle, have demonstrated that this can still be done. We refer to sections 2.1.8 and 3.3.4.7 for further considerations on consistency. The best compromise we can think of at the moment is to allow multiple levels of consistency to coexist in the database, and to let the user decide on the consistency model a per-stream basis.

There is a few recent literature with interesting approaches to state machine replication for streams, such as building the replication layer itself on top of a stream processing engine [193]. We recommend to keep track of the literature and research here, since this is a relatively young area.

5.1.2 Further Optimizations

In this final section, we outline some of the possible optimizations to Raft on the one hand, and to distributed event stores on the other. Throughout this work, we pointed to them in the discussions of consistency models, system design and our implementation.

- Sharding is essential to allow for horizontal scalability, either by time splits, on the current split, or both.
- Support of auto-scaling is crucial for production-ready systems, especially those running in containerized environments, such as Kubernetes.
- Monitoring throughput and stream convergence behaviour allows to continously calculate the maximum tolerated throughput that does not result in a negative convergence rate. Since negative convergence causes random leader elections, selected clients should receive push-backs in this case to prevent the system from becoming unavailable for all clients.
- Strong consistency is very expensive when running geo-replicated systems. We recommend to run an event processing agent between geo-replicated systems if strong consistency is not neccessary across the globe.
- There is a multitude of raft optimizations that we presented in section 2.2.13, that can increase the total throughput, but may violate the understandability property of Raft. Another paper also tries to map popular optimizations of Paxos to Raft [194].
- We use the naive implementation of the Raft log, provided by Apache Ratis, which is extremely slow. The Ratis team recommends that you implement your own log, and so do we. We proposed a log that only consists of pointers to the event store, to support the “The log is the database” philosophy again.
- We have not implemented a snapshotting mechanism for the event store state machine and therefore strongly recommend that this should be done.
- Our buffer is a simple solution, but it violates atomicity of Raft log entries. This should be done properly.
- We recommend to introduce learner roles or something similar to put load of from the leader. With increasing load, the leader is more likely to delay the delivery of heartbeats. We experienced in our evaluation, that with spikes in throughput that are above the maximum tolerated throughput, the chance of blocking all of the the leader’s threads increases. Once this happens, the leader won’t send heartbeats anymore in time, which triggers leader election. By moving the read load from the leader, the chance for this to happen decreases

at least by some degrees.

- We can also imagine a dedicated role that serves continuous queries.
- But first, distributed queries must be implemented, since our prototype does not provide query processing at all.

Appendix

ChronicleDB on a Raft: Performance Tests

In [2]:
```
import pandas as pd
import numpy as np
import altair as alt

#pip install vega
#jupyter nbextension install --sys-prefix --py vega

#alt.renderers.enable('notebook')
alt.renderers.enable('html')
```

Out[2]:
```
RendererRegistry.enable('html')
```

In [58]:
```
import requests

#host = 'http://localhost:8080/api'
host = 'http://3.121.183.166:8080/api'


def run_test(count, target='event-store'):
    query = { "batchSize": count }
    response = requests.request('GET', f'{host}/sys-info/performance/measure/{target}/insert-events/{count}', params=
    return response.json()

def clear_stream(stream='demo-event-store', target='event-store'):
    response = requests.request('DELETE', f'{host}/{target}/streams/{stream}/events')
    token = response.json()
    print(token)
    response = requests.request('POST', f'{host}/{target}/clear-request-confirmation', json=token)
    return response
```

In [59]:
```
clear_stream('demo_event_store', target='event-store/embedded')
```

```
{'streamName': 'demo_event_store', 'token': '8_1gIecvKQimY9cpEha522qhH0UwDYSn'}
```
Out[59]:
```
<Response [200]>
```

In [60]:
```
# To create streams in the embedded db
def create_schema_for_embedded_db(schema):
    response = requests.request('POST', f'{host}/event-store/embedded/streams', json=schema)
    return response
```

In [61]:
```
create_schema_for_embedded_db({
    "streamName": "demo_event_store",
    "schema": [
        {
            "name": "SYMBOL",
            "type": "STRING",
            "properties": {}
        },
        {
            "name": "SECURITYTYPE",
            "type": "INTEGER",
            "properties": {}
        },
        {
            "name": "LASTTRADEPRICE",
            "type": "FLOAT",
            "properties": {}
        }
    ]
})
```

Out[61]:
```
<Response [500]>
```

In [62]:
```
import datetime
import pandas as pd

def run_tests(counts=[1,100], trials_per_count=1, env_info={}, target='event-store', stream='demo-event-store', csv_c
    now = datetime.datetime.now().replace(microsecond=0).isoformat().replace(':', '-')
    csv_name = f'results-{now}.csv'

    env_info_columns = list(env_info.keys())

    df = pd.DataFrame(columns = [*env_info_columns, 'buffer_size_in_bytes', 'event_count', 'trial', 'duration_in_ms',

    for count in counts:
        for i in range(0, trials_per_count):
            print(f'Trial {i+1} for event count {count}')
            results = run_test(count, target)
            print(results['message'])
```

```
            df = df.append({
                **env_info,
                'buffer_size_in_bytes': results['bufferSize'],
                'trial': i+1,
                'event_count': count,
                'duration_in_ms': results['timeElapsed'],
                'measured_on': now
            }, ignore_index=True)
            df.to_csv(f'measurements/{csv_dir}/{csv_name}', index=False)
            # clean service after each trial. Must delete all events to avoid OOO
            stream_cleared = clear_stream(stream, target)
            print(stream_cleared)

    return df
```

In [14]:
```
clear_stream()
```

Out[14]:
```
{'streamName': 'demo-event-store', 'token': 'wBINNQQtadtotOFZKgo-jathuDzaJlNc'}
<Response [200]>
```

In [72]:
```
env_info = {
    #'cluster_type': 'localMacBookProIntelI9',
    'cluster_type': 'awsLightsail2GB',
    'node_count': 3,
    'event_type': 'randomized',
    'buffer_type': 'blocking'
}

run_tests(counts=[1, 100, 10000, 1000000], trials_per_count=10, env_info=env_info, target='event-store', csv_dir='aws

# run_tests(counts=[10000000], trials_per_count=10, env_info=env_info)

# an event of our examples has ~21 bytes
```

In [64]:
```
# measure performance of non-replicated, embedded store (the original one)

env_info = {
    'cluster_type': 'standalone-embedded-aws',
    'node_count': 1,
    'event_type': 'randomized',
    'buffer_type': 'none-embedded'
}

run_tests(counts=[1, 100, 10000, 1000000], trials_per_count=10, env_info=env_info, target='event-store/embedded', str
```

```
Trial 1 for event count 1
runInsertIntoEmbeddedEventStoreMeasurements 1 times in 1 batches (1 threads per batch)
Batch 1: Inserted events 1 times [0ms; ~0ms per call]
Total time: 6ms

{'streamName': 'demo_event_store', 'token': 'A7XxGuXX47Axb42h37yMl-co7PxD58As'}
<Response [200]>
Trial 2 for event count 1
runInsertIntoEmbeddedEventStoreMeasurements 1 times in 1 batches (1 threads per batch)
Batch 1: Inserted events 1 times [4ms; ~4ms per call]
Total time: 6ms

{'streamName': 'demo_event_store', 'token': 'B6-priuMGV7r3m5ZgOPRnlCjW_7z0HVa'}
<Response [200]>
Trial 3 for event count 1
runInsertIntoEmbeddedEventStoreMeasurements 1 times in 1 batches (1 threads per batch)
Batch 1: Inserted events 1 times [90ms; ~90ms per call]
Total time: 147ms

{'streamName': 'demo_event_store', 'token': 'fbeReEoSkG6RkSWSiH0vA2LLkRvGj2WW'}
<Response [200]>
Trial 4 for event count 1
runInsertIntoEmbeddedEventStoreMeasurements 1 times in 1 batches (1 threads per batch)
Batch 1: Inserted events 1 times [4ms; ~4ms per call]
Total time: 9ms

{'streamName': 'demo_event_store', 'token': 'eBYt5hZPeXzilSJGhkcLvRyH0yc_i7iI'}
<Response [200]>
Trial 5 for event count 1
runInsertIntoEmbeddedEventStoreMeasurements 1 times in 1 batches (1 threads per batch)
Batch 1: Inserted events 1 times [0ms; ~0ms per call]
Total time: 4ms

{'streamName': 'demo_event_store', 'token': 'fHqdXUGD-fFxYUSGU2HEA1Va_1a6TKzm'}
<Response [200]>
Trial 6 for event count 1
runInsertIntoEmbeddedEventStoreMeasurements 1 times in 1 batches (1 threads per batch)
Batch 1: Inserted events 1 times [2ms; ~2ms per call]
Total time: 12ms

{'streamName': 'demo_event_store', 'token': '3cz4wtQmlixYIAlzNlzi5tWROPiPojxB'}
<Response [200]>
```

```
Trial 7 for event count 1
runInsertIntoEmbeddedEventStoreMeasurements 1 times in 1 batches (1 threads per batch)
Batch 1: Inserted events 1 times [1ms; ~1ms per call]
Total time: 7ms

{'streamName': 'demo_event_store', 'token': 'yW5wBGs5b_dik8AVnaEnVdUBI53UWllO'}
<Response [200]>
Trial 8 for event count 1
runInsertIntoEmbeddedEventStoreMeasurements 1 times in 1 batches (1 threads per batch)
Batch 1: Inserted events 1 times [6ms; ~6ms per call]
Total time: 9ms

{'streamName': 'demo_event_store', 'token': 'PRrVKvdknV9D5sadmx79V6CnkkXopUH_'}
<Response [200]>
Trial 9 for event count 1
runInsertIntoEmbeddedEventStoreMeasurements 1 times in 1 batches (1 threads per batch)
Batch 1: Inserted events 1 times [0ms; ~0ms per call]
Total time: 2ms

{'streamName': 'demo_event_store', 'token': 'V6Qt8Za4MRvU0svboZ__9JVK3v5zSsf7'}
<Response [200]>
Trial 10 for event count 1
runInsertIntoEmbeddedEventStoreMeasurements 1 times in 1 batches (1 threads per batch)
Batch 1: Inserted events 1 times [0ms; ~0ms per call]
Total time: 0ms

{'streamName': 'demo_event_store', 'token': '6xBF2V86ZWhZvHiBmKI1mq89gHskamX0'}
<Response [200]>
Trial 1 for event count 100
runInsertIntoEmbeddedEventStoreMeasurements 100 times in 1 batches (100 threads per batch)
Batch 1: Inserted events 100 times [84ms; ~0ms per call]
Total time: 86ms

{'streamName': 'demo_event_store', 'token': 'Fm_-hAGPMMXPHduhGy-YoiFxObIy7CE0'}
<Response [200]>
Trial 2 for event count 100
runInsertIntoEmbeddedEventStoreMeasurements 100 times in 1 batches (100 threads per batch)
Batch 1: Inserted events 100 times [0ms; ~0ms per call]
Total time: 3ms

{'streamName': 'demo_event_store', 'token': 'z2ccPLCFhjM9agEDidsbGYMG_kehMU7y'}
<Response [200]>
Trial 3 for event count 100
runInsertIntoEmbeddedEventStoreMeasurements 100 times in 1 batches (100 threads per batch)
Batch 1: Inserted events 100 times [3ms; ~0ms per call]
Total time: 3ms

{'streamName': 'demo_event_store', 'token': 'Ly0S72KhH2mGWPuiUIoq1M8TVa9Fit6y'}
<Response [200]>
Trial 4 for event count 100
runInsertIntoEmbeddedEventStoreMeasurements 100 times in 1 batches (100 threads per batch)
Batch 1: Inserted events 100 times [1ms; ~0ms per call]
Total time: 22ms

{'streamName': 'demo_event_store', 'token': '4oVGbq7pGBVc7rrSnZiuZBKQVDZc0B-z'}
<Response [200]>
Trial 5 for event count 100
runInsertIntoEmbeddedEventStoreMeasurements 100 times in 1 batches (100 threads per batch)
Batch 1: Inserted events 100 times [9ms; ~0ms per call]
Total time: 15ms

{'streamName': 'demo_event_store', 'token': 'pII84qkdQ5Ty1bmToFl-iecWmkw9GnjG'}
<Response [200]>
Trial 6 for event count 100
runInsertIntoEmbeddedEventStoreMeasurements 100 times in 1 batches (100 threads per batch)
Batch 1: Inserted events 100 times [0ms; ~0ms per call]
Total time: 3ms

{'streamName': 'demo_event_store', 'token': 'cZpJjIu6gQfWmmHsrUr08rKXTMP2haX8'}
<Response [200]>
Trial 7 for event count 100
runInsertIntoEmbeddedEventStoreMeasurements 100 times in 1 batches (100 threads per batch)
Batch 1: Inserted events 100 times [1ms; ~0ms per call]
Total time: 3ms

{'streamName': 'demo_event_store', 'token': 'BGPUkYdolR43y-C9DjaYOtPBzZjOk7Fl'}
<Response [200]>
Trial 8 for event count 100
runInsertIntoEmbeddedEventStoreMeasurements 100 times in 1 batches (100 threads per batch)
Batch 1: Inserted events 100 times [3ms; ~0ms per call]
Total time: 9ms

{'streamName': 'demo_event_store', 'token': '7gLvBL8djHFB9ggLHv_CLfFaSVLcuRuM'}
<Response [200]>
Trial 9 for event count 100
runInsertIntoEmbeddedEventStoreMeasurements 100 times in 1 batches (100 threads per batch)
Batch 1: Inserted events 100 times [4ms; ~0ms per call]
Total time: 6ms

{'streamName': 'demo_event_store', 'token': '9W7VvlPSgVSfH8W1Spf7rs_UUn19BqMU'}
<Response [200]>
Trial 10 for event count 100
```

```
runInsertIntoEmbeddedEventStoreMeasurements 100 times in 1 batches (100 threads per batch)
Batch 1: Inserted events 100 times [4ms; ~0ms per call]
Total time: 26ms

{'streamName': 'demo_event_store', 'token': '1P5TYVSKJLJJ7Na7MnLY0m4bXwbf41Fy'}
<Response [200]>
Trial 1 for event count 10000
runInsertIntoEmbeddedEventStoreMeasurements 10000 times in 1 batches (10000 threads per batch)
Batch 1: Inserted events 10000 times [42ms; ~0ms per call]
Total time: 60ms

{'streamName': 'demo_event_store', 'token': 'G6MWq1xwV4TIc06iCqYWdMFpsu0WI5cb'}
<Response [200]>
Trial 2 for event count 10000
runInsertIntoEmbeddedEventStoreMeasurements 10000 times in 1 batches (10000 threads per batch)
Batch 1: Inserted events 10000 times [31ms; ~0ms per call]
Total time: 34ms

{'streamName': 'demo_event_store', 'token': 'duJInbtoYp3s3Ijbo6ko4g2B2VJVXL7o'}
<Response [200]>
Trial 3 for event count 10000
runInsertIntoEmbeddedEventStoreMeasurements 10000 times in 1 batches (10000 threads per batch)
Batch 1: Inserted events 10000 times [20ms; ~0ms per call]
Total time: 23ms

{'streamName': 'demo_event_store', 'token': 'Hmj4RASlr4Xnd2xAR5ZBkK4FtnhhVYks'}
<Response [200]>
Trial 4 for event count 10000
runInsertIntoEmbeddedEventStoreMeasurements 10000 times in 1 batches (10000 threads per batch)
Batch 1: Inserted events 10000 times [25ms; ~0ms per call]
Total time: 28ms

{'streamName': 'demo_event_store', 'token': 'qEKg5bFmkvVQER0kO-E3nQhbsTJZhDAd'}
<Response [200]>
Trial 5 for event count 10000
runInsertIntoEmbeddedEventStoreMeasurements 10000 times in 1 batches (10000 threads per batch)
Batch 1: Inserted events 10000 times [6ms; ~0ms per call]
Total time: 9ms

{'streamName': 'demo_event_store', 'token': '-5dpMzrBmrHZktoGeh9C2-9rPbNsDdSn'}
<Response [200]>
Trial 6 for event count 10000
runInsertIntoEmbeddedEventStoreMeasurements 10000 times in 1 batches (10000 threads per batch)
Batch 1: Inserted events 10000 times [14ms; ~0ms per call]
Total time: 15ms

{'streamName': 'demo_event_store', 'token': 'oCZ7t_FcHR15FH9zO8DqIEvj_kTY_AQn'}
<Response [200]>
Trial 7 for event count 10000
runInsertIntoEmbeddedEventStoreMeasurements 10000 times in 1 batches (10000 threads per batch)
Batch 1: Inserted events 10000 times [13ms; ~0ms per call]
Total time: 19ms

{'streamName': 'demo_event_store', 'token': 'v3tfDD0keA6clEoVAkSW3m74mjsvIlY5'}
<Response [200]>
Trial 8 for event count 10000
runInsertIntoEmbeddedEventStoreMeasurements 10000 times in 1 batches (10000 threads per batch)
Batch 1: Inserted events 10000 times [101ms; ~0ms per call]
Total time: 141ms

{'streamName': 'demo_event_store', 'token': 'rLyY979qzGBMCQP9BefbliyAE43baiQI'}
<Response [200]>
Trial 9 for event count 10000
runInsertIntoEmbeddedEventStoreMeasurements 10000 times in 1 batches (10000 threads per batch)
Batch 1: Inserted events 10000 times [14ms; ~0ms per call]
Total time: 15ms

{'streamName': 'demo_event_store', 'token': 'i-iXPJKllQR44Ej4ocVloZu9l0nzDe1c'}
<Response [200]>
Trial 10 for event count 10000
runInsertIntoEmbeddedEventStoreMeasurements 10000 times in 1 batches (10000 threads per batch)
Batch 1: Inserted events 10000 times [16ms; ~0ms per call]
Total time: 19ms

{'streamName': 'demo_event_store', 'token': 'lePsZWV8EyRZJmvuFZ7qUaO16Uu8ZYMV'}
<Response [200]>
Trial 1 for event count 1000000
runInsertIntoEmbeddedEventStoreMeasurements 1000000 times in 1 batches (1000000 threads per batch)
Batch 1: Inserted events 1000000 times [580ms; ~0ms per call]
Total time: 580ms

{'streamName': 'demo_event_store', 'token': 'H3PP44X3q2wd5AP4DCKwvuV69Epo6tua'}
<Response [200]>
Trial 2 for event count 1000000
runInsertIntoEmbeddedEventStoreMeasurements 1000000 times in 1 batches (1000000 threads per batch)
Batch 1: Inserted events 1000000 times [422ms; ~0ms per call]
Total time: 422ms

{'streamName': 'demo_event_store', 'token': 'HO1nrFI0pXo8RcgFjKV31nWQq6OhvYLU'}
<Response [200]>
Trial 3 for event count 1000000
runInsertIntoEmbeddedEventStoreMeasurements 1000000 times in 1 batches (1000000 threads per batch)
```

```
Batch 1: Inserted events 1000000 times [552ms; ~0ms per call]
Total time: 554ms

{'streamName': 'demo_event_store', 'token': '6NKz3uKXzvazqEhX7wQhlOZe4fy3MG8h'}
<Response [200]>
Trial 4 for event count 1000000
runInsertIntoEmbeddedEventStoreMeasurements 1000000 times in 1 batches (1000000 threads per batch)
Batch 1: Inserted events 1000000 times [460ms; ~0ms per call]
Total time: 460ms

{'streamName': 'demo_event_store', 'token': 'v4UJxLgAmH5KeKOyerHDbQf5jrIeWY8p'}
<Response [200]>
Trial 5 for event count 1000000
runInsertIntoEmbeddedEventStoreMeasurements 1000000 times in 1 batches (1000000 threads per batch)
Batch 1: Inserted events 1000000 times [437ms; ~0ms per call]
Total time: 440ms

{'streamName': 'demo_event_store', 'token': 'TRmBkm9aYXTRNL8a14WzUp_1WD0yivyd'}
<Response [200]>
Trial 6 for event count 1000000
runInsertIntoEmbeddedEventStoreMeasurements 1000000 times in 1 batches (1000000 threads per batch)
Batch 1: Inserted events 1000000 times [398ms; ~0ms per call]
Total time: 398ms

{'streamName': 'demo_event_store', 'token': 'RWlPCc0NgArjKG4oNudJ5AGyag7gmBYF'}
<Response [200]>
Trial 7 for event count 1000000
runInsertIntoEmbeddedEventStoreMeasurements 1000000 times in 1 batches (1000000 threads per batch)
Batch 1: Inserted events 1000000 times [500ms; ~0ms per call]
Total time: 507ms

{'streamName': 'demo_event_store', 'token': 'RV9PKk86TaM66RJXKCiNt0lg-6pocLVN'}
<Response [200]>
Trial 8 for event count 1000000
runInsertIntoEmbeddedEventStoreMeasurements 1000000 times in 1 batches (1000000 threads per batch)
Batch 1: Inserted events 1000000 times [388ms; ~0ms per call]
Total time: 388ms

{'streamName': 'demo_event_store', 'token': 'liJdttlFeGEK0qCrMXr_ydI9D3zwNruH'}
<Response [200]>
Trial 9 for event count 1000000
runInsertIntoEmbeddedEventStoreMeasurements 1000000 times in 1 batches (1000000 threads per batch)
Batch 1: Inserted events 1000000 times [439ms; ~0ms per call]
Total time: 440ms

{'streamName': 'demo_event_store', 'token': 'BwZRbMEBnXi1L-FWftm_5fgRqfdne9Xz'}
<Response [200]>
Trial 10 for event count 1000000
runInsertIntoEmbeddedEventStoreMeasurements 1000000 times in 1 batches (1000000 threads per batch)
Batch 1: Inserted events 1000000 times [574ms; ~0ms per call]
Total time: 582ms

{'streamName': 'demo_event_store', 'token': 'QHZfLbZMo_cOzuHGZ81OUbTsWlJoBAt1'}
<Response [200]>
```

Out[64]:

	cluster_type	node_count	event_type	buffer_type	buffer_size_in_bytes	event_count	trial	duration_in_ms	measured_on
0	standalone-embedded-aws	1	randomized	none-embedded	0	1	1	6	2022-03-16T15-44-04
1	standalone-embedded-aws	1	randomized	none-embedded	0	1	2	6	2022-03-16T15-44-04
2	standalone-embedded-aws	1	randomized	none-embedded	0	1	3	147	2022-03-16T15-44-04
3	standalone-embedded-aws	1	randomized	none-embedded	0	1	4	9	2022-03-16T15-44-04
4	standalone-embedded-aws	1	randomized	none-embedded	0	1	5	4	2022-03-16T15-44-04
5	standalone-embedded-aws	1	randomized	none-embedded	0	1	6	12	2022-03-16T15-44-04
6	standalone-embedded-aws	1	randomized	none-embedded	0	1	7	7	2022-03-16T15-44-04
7	standalone-embedded-aws	1	randomized	none-embedded	0	1	8	9	2022-03-16T15-44-04
8	standalone-embedded-aws	1	randomized	none-embedded	0	1	9	2	2022-03-16T15-44-04
9	standalone-embedded-aws	1	randomized	none-embedded	0	1	10	0	2022-03-16T15-44-04
10	standalone-embedded-aws	1	randomized	none-embedded	0	100	1	86	2022-03-16T15-44-04
11	standalone-embedded-aws	1	randomized	none-embedded	0	100	2	3	2022-03-16T15-44-04
12	standalone-embedded-aws	1	randomized	none-embedded	0	100	3	3	2022-03-16T15-44-04
13	standalone-embedded-aws	1	randomized	none-embedded	0	100	4	22	2022-03-16T15-44-04
14	standalone-embedded-aws	1	randomized	none-embedded	0	100	5	15	2022-03-16T15-44-04
15	standalone-embedded-aws	1	randomized	none-embedded	0	100	6	3	2022-03-16T15-44-04
16	standalone-embedded-aws	1	randomized	none-embedded	0	100	7	3	2022-03-16T15-44-04
17	standalone-embedded-aws	1	randomized	none-embedded	0	100	8	9	2022-03-16T15-44-04
18	standalone-embedded-aws	1	randomized	none-embedded	0	100	9	6	2022-03-16T15-44-04
19	standalone-embedded-aws	1	randomized	none-embedded	0	100	10	26	2022-03-16T15-44-04
20	standalone-embedded-aws	1	randomized	none-embedded	0	10000	1	60	2022-03-16T15-44-04

	cluster_type	node_count	event_type	buffer_type	buffer_size_in_bytes	event_count	trial	duration_in_ms	measured_on
21	standalone-embedded-aws	1	randomized	none-embedded	0	10000	2	34	2022-03-16T15-44-04
22	standalone-embedded-aws	1	randomized	none-embedded	0	10000	3	23	2022-03-16T15-44-04
23	standalone-embedded-aws	1	randomized	none-embedded	0	10000	4	28	2022-03-16T15-44-04
24	standalone-embedded-aws	1	randomized	none-embedded	0	10000	5	9	2022-03-16T15-44-04
25	standalone-embedded-aws	1	randomized	none-embedded	0	10000	6	15	2022-03-16T15-44-04
26	standalone-embedded-aws	1	randomized	none-embedded	0	10000	7	19	2022-03-16T15-44-04
27	standalone-embedded-aws	1	randomized	none-embedded	0	10000	8	141	2022-03-16T15-44-04
28	standalone-embedded-aws	1	randomized	none-embedded	0	10000	9	15	2022-03-16T15-44-04
29	standalone-embedded-aws	1	randomized	none-embedded	0	10000	10	19	2022-03-16T15-44-04
30	standalone-embedded-aws	1	randomized	none-embedded	0	1000000	1	580	2022-03-16T15-44-04
31	standalone-embedded-aws	1	randomized	none-embedded	0	1000000	2	422	2022-03-16T15-44-04
32	standalone-embedded-aws	1	randomized	none-embedded	0	1000000	3	554	2022-03-16T15-44-04
33	standalone-embedded-aws	1	randomized	none-embedded	0	1000000	4	460	2022-03-16T15-44-04
34	standalone-embedded-aws	1	randomized	none-embedded	0	1000000	5	440	2022-03-16T15-44-04
35	standalone-embedded-aws	1	randomized	none-embedded	0	1000000	6	398	2022-03-16T15-44-04
36	standalone-embedded-aws	1	randomized	none-embedded	0	1000000	7	507	2022-03-16T15-44-04
37	standalone-embedded-aws	1	randomized	none-embedded	0	1000000	8	388	2022-03-16T15-44-04
38	standalone-embedded-aws	1	randomized	none-embedded	0	1000000	9	440	2022-03-16T15-44-04
39	standalone-embedded-aws	1	randomized	none-embedded	0	1000000	10	582	2022-03-16T15-44-04

Evaluating the benchmark results

All of this is currently under conditions without out-of-order events

In [1]:
```
# load all measurements

from os import walk

resource_folder = 'measurements'

resources = next(walk(resource_folder), (None, None, []))[2]

print(f'{len(resources)} files in total')
```

```
24 files in total
```

In [3]:
```
datasets = {filename: pd.read_csv(f'{resource_folder}/{filename}', sep=',', encoding='utf-8', error_bad_lines=False)
```

In [4]:
```
all_measurements_df = pd.concat(datasets, ignore_index=True)

all_measurements_df
```

Out[4]:

	cluster_type	node_count	event_type	buffer_type	buffer_size_in_bytes	event_count	trial	duration_in_ms	measured_on
0	local17	3	randomized	blocking	1048576	10000000	1	124276	2022-01-29T15-16-22
1	local17	3	randomized	blocking	1048576	10000000	2	122494	2022-01-29T15-16-22
2	local17	3	randomized	blocking	1048576	10000000	3	117724	2022-01-29T15-16-22
3	local17	3	randomized	blocking	1048576	10000000	4	119596	2022-01-29T15-16-22
4	local17	3	randomized	blocking	1048576	10000000	5	114629	2022-01-29T15-16-22
...	...	...	...	...	...	...	...	...	...
830	local17	3	randomized	blocking	10240	1000000	6	20389	2022-01-29T12-48-43
831	local17	3	randomized	blocking	10240	1000000	7	20635	2022-01-29T12-48-43
832	local17	3	randomized	blocking	10240	1000000	8	21030	2022-01-29T12-48-43
833	local17	3	randomized	blocking	10240	1000000	9	21108	2022-01-29T12-48-43
834	local17	3	randomized	blocking	10240	1000000	10	21375	2022-01-29T12-48-43

835 rows × 9 columns

In [5]:
```
# supress warnings
pd.options.mode.chained_assignment = None

# remove rows with duration = 0 to avoid divide by zero
all_measurements_df = all_measurements_df[all_measurements_df['duration_in_ms'] > 0]
```

```
# replace buffer_size 0 with 1 to allow log scale plotting
all_measurements_df.loc[all_measurements_df['buffer_size_in_bytes'] == 0, 'buffer_size_in_bytes'] = 1

all_measurements_df['duration_per_event'] = all_measurements_df['duration_in_ms'] / all_measurements_df['event_count'
all_measurements_df['duration_per_mio_events'] = all_measurements_df['duration_per_event'] * 1000000
all_measurements_df['duration_per_mio_events_in_sec'] = all_measurements_df['duration_per_mio_events'] / 1000
all_measurements_df['events_per_second'] = 1000 / all_measurements_df['duration_per_event']

pd.options.mode.chained_assignment = 'warn'

all_measurements_df.describe()
```

Out[5]:

	node_count	buffer_size_in_bytes	event_count	trial	duration_in_ms	duration_per_event	duration_per_mio_events	duration_per_mio_eve
count	743.000000	7.430000e+02	7.430000e+02	743.000000	743.000000	743.000000	7.430000e+02	7
mean	3.189771	6.178415e+05	3.998901e+05	5.425303	14820.282638	2.483687	2.483687e+06	24
std	1.155231	5.059331e+05	1.204331e+06	2.891013	61681.256728	12.500844	1.250084e+07	125
min	1.000000	1.000000e+00	1.000000e+00	1.000000	1.000000	0.000388	3.880000e+02	
25%	3.000000	1.024000e+04	1.000000e+02	3.000000	4.000000	0.008908	8.908000e+03	
50%	3.000000	1.048576e+06	1.000000e+04	5.000000	37.000000	0.020000	2.000000e+04	
75%	3.000000	1.048576e+06	1.000000e+06	8.000000	7308.000000	0.225000	2.250000e+05	2
max	6.000000	1.048576e+06	1.000000e+07	10.000000	485271.000000	246.000000	2.460000e+08	2460

In [6]:

```
#replicated_measurements_df = all_measurements_df[all_measurements_df['cluster_type'] != 'standalone-embedded']
replicated_measurements_df = all_measurements_df[all_measurements_df['cluster_type'] == 'local17']
replicated_measurements_3_nodes_df = replicated_measurements_df[replicated_measurements_df['node_count'] == 3]
embedded_measurements_df = all_measurements_df[all_measurements_df['cluster_type'] == 'standalone-embedded']
remote_replicated_measurements_df = all_measurements_df[all_measurements_df['cluster_type'] == 'awsLightsail2GB']

local_and_remote_replicated_measurements_3_nodes_df = all_measurements_df[all_measurements_df['node_count'] == 3]
```

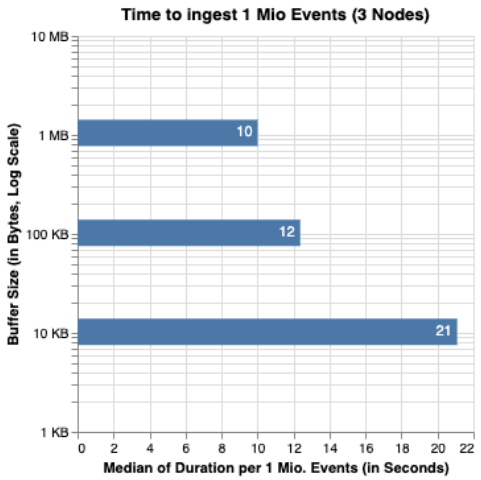
In [26]:

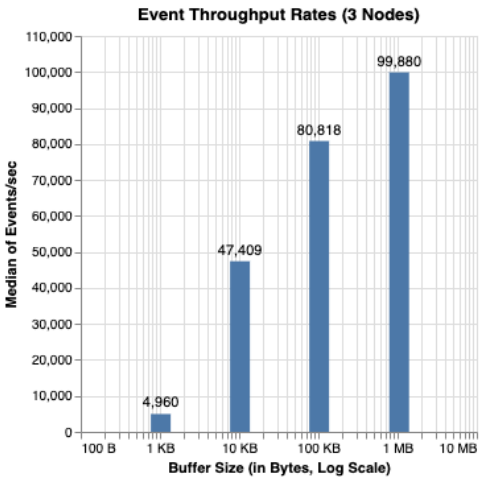
```
import altair as alt

def plot_event_rate(df, title="Event Throughput Rates", show_benchmark=False, show_benchmark_at=10000000, aggregate='
    buffer_size_label_expr = "datum.label && datum.label[0] == '1' ? (datum.value >= 1000000 ? datum.value / 1000000
    if show_benchmark:
        buffer_size_label_expr = f'datum.value == {show_benchmark_at} ? "Standalone" : ({buffer_size_label_expr})'

    blocking_queue_event_rate_plot = alt.Chart(df).mark_bar(clip=True, width=15).encode(
        x=alt.X('buffer_size_in_bytes:Q', scale=alt.Scale(type='log'), title="Buffer Size (in Bytes, Log Scale)", axi
            labelExpr=buffer_size_label_expr
        )),
        y=alt.Y(f'{aggregate}(events_per_second):Q', title=f"{aggregate.capitalize()} of Events/sec", scale=alt.Scale
        color=alt.condition(
            alt.datum.buffer_size_in_bytes == show_benchmark_at,
            alt.value('#919eac'),
            alt.value('#4c78a8')
        )
    ).properties(width=300, height=300, title=title)

    return blocking_queue_event_rate_plot + blocking_queue_event_rate_plot.mark_text(
        align='center',
        color='black',
        dx=0,
        dy=-8
    ).encode(
        text=alt.Text(f'{aggregate}(events_per_second):Q', format=',.0f'),
        color=alt.value('black')
    ).transform_calculate(label='datum.y + " inches"')
```

In [27]:

```
buffer_size_label_expr = "datum.label && datum.label[0] == '1' ? (datum.value >= 1000000 ? datum.value / 1000000 + '

duration_plot = alt.Chart(replicated_measurements_3_nodes_df[replicated_measurements_3_nodes_df['buffer_size_in_bytes
    x=alt.X('median(duration_per_mio_events_in_sec):Q', title="Median of Duration per 1 Mio. Events (in Seconds)"),
    y=alt.Y('buffer_size_in_bytes:Q', scale=alt.Scale(type='log'), title="Buffer Size (in Bytes, Log Scale)", axis=al
        labelExpr=buffer_size_label_expr
    )),
).properties(width=300, title="Time to ingest 1 Mio Events (3 Nodes)")

duration_plot = duration_plot + duration_plot.mark_text(
    align='right',
    color='white',
    dx=-4,
    #dy=-18
).encode(
    text=alt.Text('median(duration_per_mio_events_in_sec):Q', format=',.0f'),
)

duration_plot | plot_event_rate(replicated_measurements_3_nodes_df[replicated_measurements_3_nodes_df['buffer_size_in
```

Out[27]:

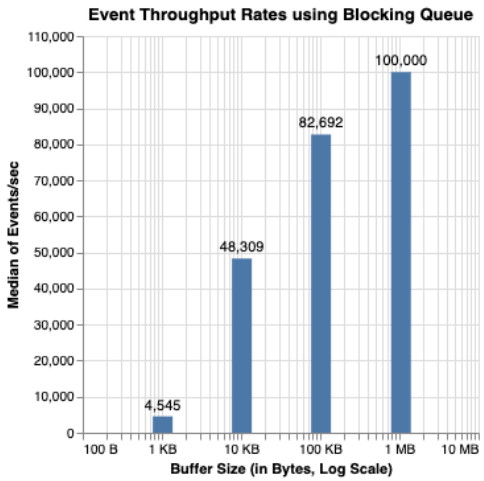


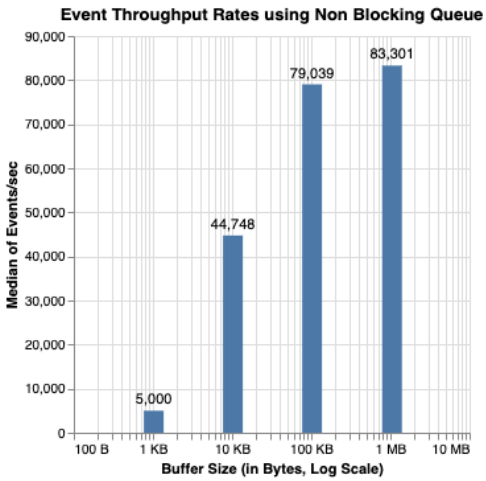


In [28]:

```
blocking_queue_df = replicated_measurements_3_nodes_df[replicated_measurements_3_nodes_df['buffer_type'] == 'blocking
non_blocking_queue_df = replicated_measurements_3_nodes_df[replicated_measurements_3_nodes_df['buffer_type'] == 'non-

plot_event_rate(blocking_queue_df, title="Event Throughput Rates using Blocking Queue") | plot_event_rate(non_blockin
```

Out[28]:

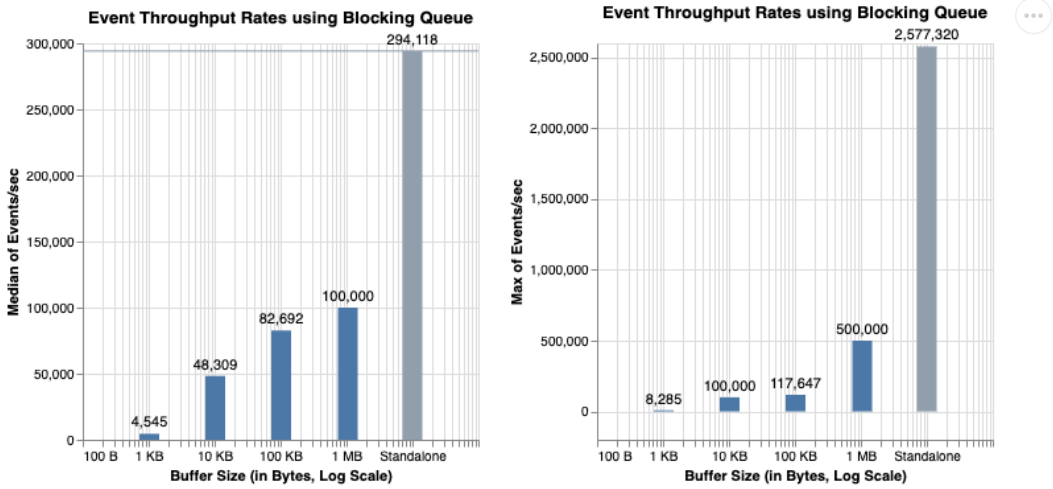




In [29]:

```
blocking_queue_vs_standalone_df = all_measurements_df[((all_measurements_df['buffer_type'] == 'blocking') & (all_meas

show_benchmark_at = 10000000

blocking_queue_vs_standalone_df.loc[(blocking_queue_vs_standalone_df['buffer_type'] == 'none-embedded'), 'buffer_size

median_plot = plot_event_rate(blocking_queue_vs_standalone_df, title="Event Throughput Rates using Blocking Queue", s
max_plot = plot_event_rate(blocking_queue_vs_standalone_df, title="Event Throughput Rates using Blocking Queue", show

benchmark_score = all_measurements_df[all_measurements_df['buffer_type'] == 'none-embedded']['events_per_second'].med

median_plot + alt.Chart(pd.DataFrame({'y': [benchmark_score]})).mark_rule(color="#919eac").encode(y='y') | max_plot

# a = plot_event_rate(all_measurements_df[(all_measurements_df['buffer_type'] == 'blocking')], title="Event Throughpu
# b = plot_event_rate(all_measurements_df[(all_measurements_df['buffer_type'] == 'none-embedded')], title="Event Throu

# a + b
```

Out[29]:



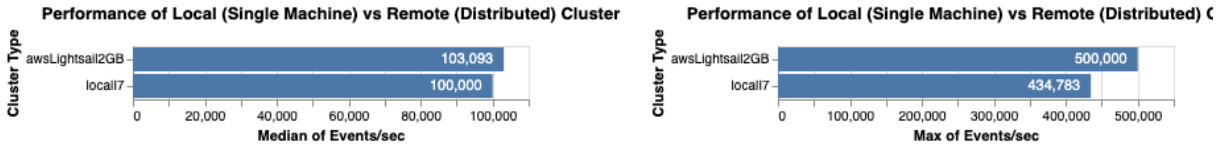
Comparison of Local Cluster (Single Machine) vs. Remote Cluster on AWS

In [11]:
```
local_vs_remote_df = local_and_remote_replicated_measurements_3_nodes_df[
    (local_and_remote_replicated_measurements_3_nodes_df['buffer_size_in_bytes'] == 1048576)
    & (local_and_remote_replicated_measurements_3_nodes_df['buffer_type'] == 'blocking')]
```

In [12]:
```
def compare_local_vs_remote_plot(aggregate='median', max_scale=550000):
    plot = alt.Chart(local_vs_remote_df).mark_bar(clip=True).encode(
        x=alt.X(f'{aggregate}(events_per_second):Q', title=f'{aggregate.capitalize()} of Events/sec', scale=alt.Scale
        y=alt.Y('cluster_type:N', title="Cluster Type"),
    ).properties(width=300, title="Performance of Local (Single Machine) vs Remote (Distributed) Cluster")

    plot = plot + plot.mark_text(
        align='right',
        color='white',
        dx=-8,
        dy=0
    ).encode(
        text=alt.Text(f'{aggregate}(events_per_second):Q', format=',.0f'),
    )

    return plot
```

In [13]:
```
compare_local_vs_remote_plot(aggregate='median', max_scale=110000) | compare_local_vs_remote_plot(aggregate='max', ma
```

Out[13]:



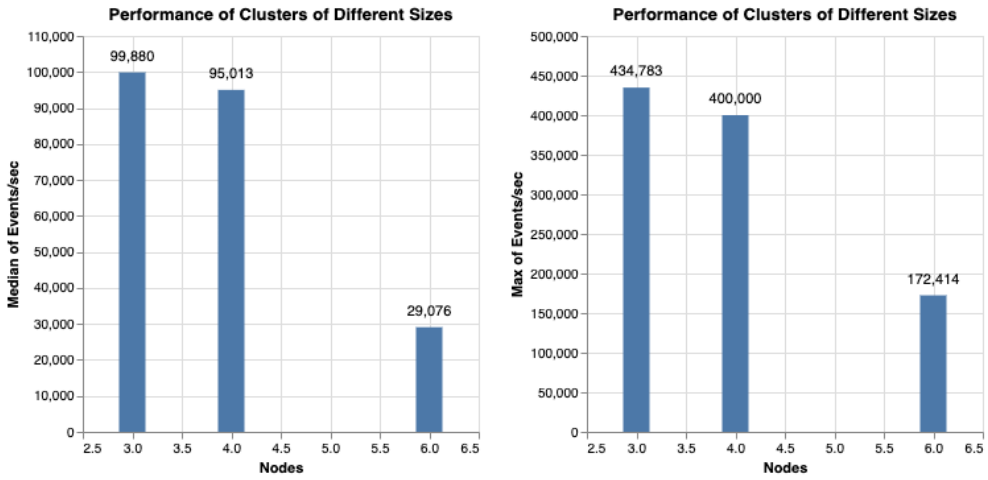
Comparison of Different Cluster Sizes (# of Nodes)

This evaluation has been run on a local machine only. We expect better results on a real cluster, since it scales out and avoids I/O bottleneck on the single machine.

In [14]:
```
def compare_nodes_plot(aggregate='median', max_scale=550000):
    node_compare_plot = alt.Chart(replicated_measurements_df[replicated_measurements_df['buffer_size_in_bytes'] == 10
        y=alt.Y(f'{aggregate}(events_per_second):Q', title=f'{aggregate.capitalize()} of Events/sec', scale=alt.Scale
        x=alt.X('node_count:Q', title="Nodes"),
    ).properties(width=300, title="Performance of Clusters of Different Sizes")

    node_compare_plot = node_compare_plot + node_compare_plot.mark_text(
        align='center',
        color='black',
        dx=0,
        dy=-12
    ).encode(
        text=alt.Text(f'{aggregate}(events_per_second):Q', format=',.0f'),
    )

    return node_compare_plot
```

In [15]:
```
compare_nodes_plot(aggregate='median', max_scale=110000) | compare_nodes_plot(aggregate='max', max_scale=500000)
```

Out[15]:



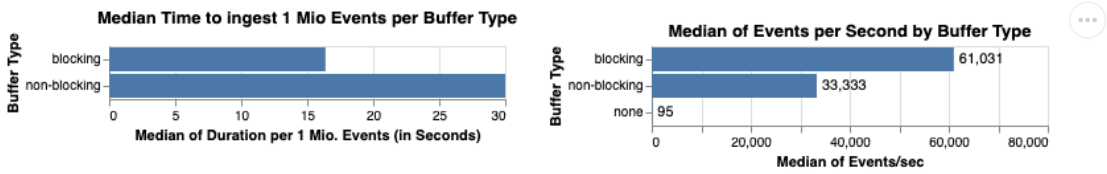
Comparison of Blocking vs Non-Blocking Buffer

We compared our two buffer implementations to decide for the best one.

In [16]:

```
buffered_measurements_df = replicated_measurements_df[replicated_measurements_df['buffer_type'] != 'none']

time_per_buffer_plot = alt.Chart(buffered_measurements_df).mark_bar(clip=True).encode(
    x=alt.X('median(duration_per_mio_events_in_sec):Q', title="Median of Duration per 1 Mio. Events (in Seconds)"),
    y=alt.Y('buffer_type:N',title="Buffer Type"),
).properties(width=300, title="Median Time to ingest 1 Mio Events per Buffer Type")

events_per_sec_plot = alt.Chart(replicated_measurements_df).mark_bar(clip=True).encode(
    x=alt.X('median(events_per_second):Q', title="Median of Events/sec", scale=alt.Scale(domain=[0, 80000])),
    y=alt.Y('buffer_type:N',title="Buffer Type"),
).properties(width=300, title="Median of Events per Second by Buffer Type")


events_per_sec_plot = events_per_sec_plot + events_per_sec_plot.mark_text(
    align='left',
    color='black',
    dx=4,
    #dy=-18
).encode(
    text=alt.Text('median(events_per_second):Q', format=',.0f'),
)

time_per_buffer_plot | events_per_sec_plot
```

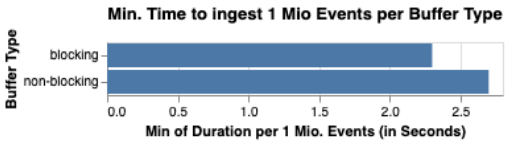
Out[16]:

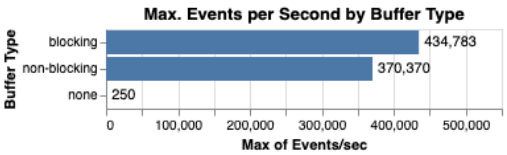


In [17]:

```
time_per_buffer_plot = alt.Chart(buffered_measurements_df).mark_bar(clip=True).encode(
    x=alt.X('min(duration_per_mio_events_in_sec):Q', title="Min of Duration per 1 Mio. Events (in Seconds)"),
    y=alt.Y('buffer_type:N',title="Buffer Type"),
).properties(width=300, title="Min. Time to ingest 1 Mio Events per Buffer Type")

events_per_sec_plot = alt.Chart(replicated_measurements_df).mark_bar(clip=True).encode(
    x=alt.X('max(events_per_second):Q', title="Max of Events/sec", scale=alt.Scale(domain=[0, 550000])),
    y=alt.Y('buffer_type:N',title="Buffer Type"),
).properties(width=300, title="Max. Events per Second by Buffer Type")


events_per_sec_plot = events_per_sec_plot + events_per_sec_plot.mark_text(
    align='left',
    color='black',
    dx=4,
    #dy=-18
).encode(
    text=alt.Text('max(events_per_second):Q', format=',.0f'),
)

time_per_buffer_plot | events_per_sec_plot
```

Out[17]:

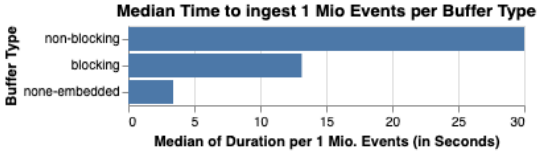


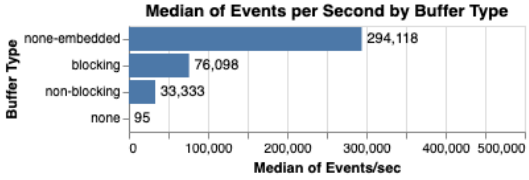


Comparison with Standalone/Embedded ChronicleDB

In [18]:
```
buffered_and_embedded_measurements_df = all_measurements_df[all_measurements_df['buffer_type'] != 'none']

time_per_buffer_plot = alt.Chart(buffered_and_embedded_measurements_df).mark_bar(clip=True).encode(
    x=alt.X('median(duration_per_mio_events_in_sec):Q', title="Median of Duration per 1 Mio. Events (in Seconds)"),
    y=alt.Y('buffer_type:N',title="Buffer Type", sort='-x'),
).properties(width=300, title="Median Time to ingest 1 Mio Events per Buffer Type")

events_per_sec_plot = alt.Chart(all_measurements_df).mark_bar(clip=True).encode(
    x=alt.X('median(events_per_second):Q', title="Median of Events/sec", scale=alt.Scale(domain=[0, 500000])),
    y=alt.Y('buffer_type:N',title="Buffer Type", sort='-x'),
).properties(width=300, title="Median of Events per Second by Buffer Type")


events_per_sec_plot = events_per_sec_plot + events_per_sec_plot.mark_text(
    align='left',
    color='black',
    dx=4,
    #dy=-18
).encode(
    text=alt.Text('median(events_per_second):Q', format=',.0f'),
)

time_per_buffer_plot | events_per_sec_plot
```

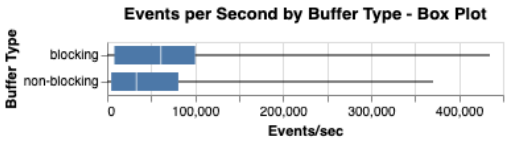
Out[18]:

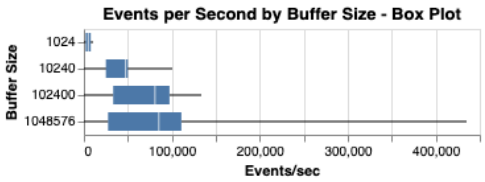




In [19]:
```
alt.Chart(buffered_measurements_df).mark_boxplot(extent='min-max').encode(
    y=alt.Y('buffer_type:N',title="Buffer Type"),
    x=alt.X('events_per_second:Q', title="Events/sec"),
).properties(width=300, title="Events per Second by Buffer Type - Box Plot") | alt.Chart(buffered_measurements_df).ma
    y=alt.Y('buffer_size_in_bytes:O',title="Buffer Size"),
    x=alt.X('events_per_second:Q', title="Events/sec"),
).properties(width=300, title="Events per Second by Buffer Size - Box Plot")
```

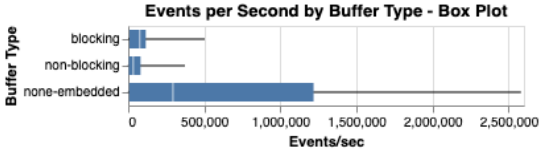
Out[19]:

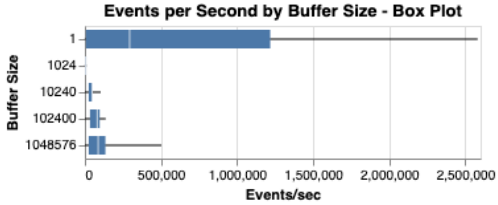




In [20]:
```
alt.Chart(buffered_and_embedded_measurements_df).mark_boxplot(extent='min-max').encode(
    y=alt.Y('buffer_type:N',title="Buffer Type"),
    x=alt.X('events_per_second:Q', title="Events/sec"),
).properties(width=300, title="Events per Second by Buffer Type - Box Plot") | alt.Chart(buffered_and_embedded_measur
    y=alt.Y('buffer_size_in_bytes:O',title="Buffer Size"),
    x=alt.X('events_per_second:Q', title="Events/sec"),
).properties(width=300, title="Events per Second by Buffer Size - Box Plot")
```

Out[20]:

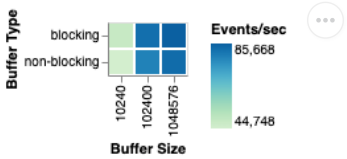




In [21]:
```
alt.Chart(replicated_measurements_df[replicated_measurements_df['buffer_size_in_bytes'] > 1024]).mark_bar(clip=True).
    color=alt.Color('median(events_per_second):Q', title="Events/sec", scale=alt.Scale(scheme='greenblue')),
    x=alt.X('buffer_size_in_bytes:N',title="Buffer Size"),
```

```
    y=alt.Y('buffer_type:N',title="Buffer Type"),
).properties(title="")
```

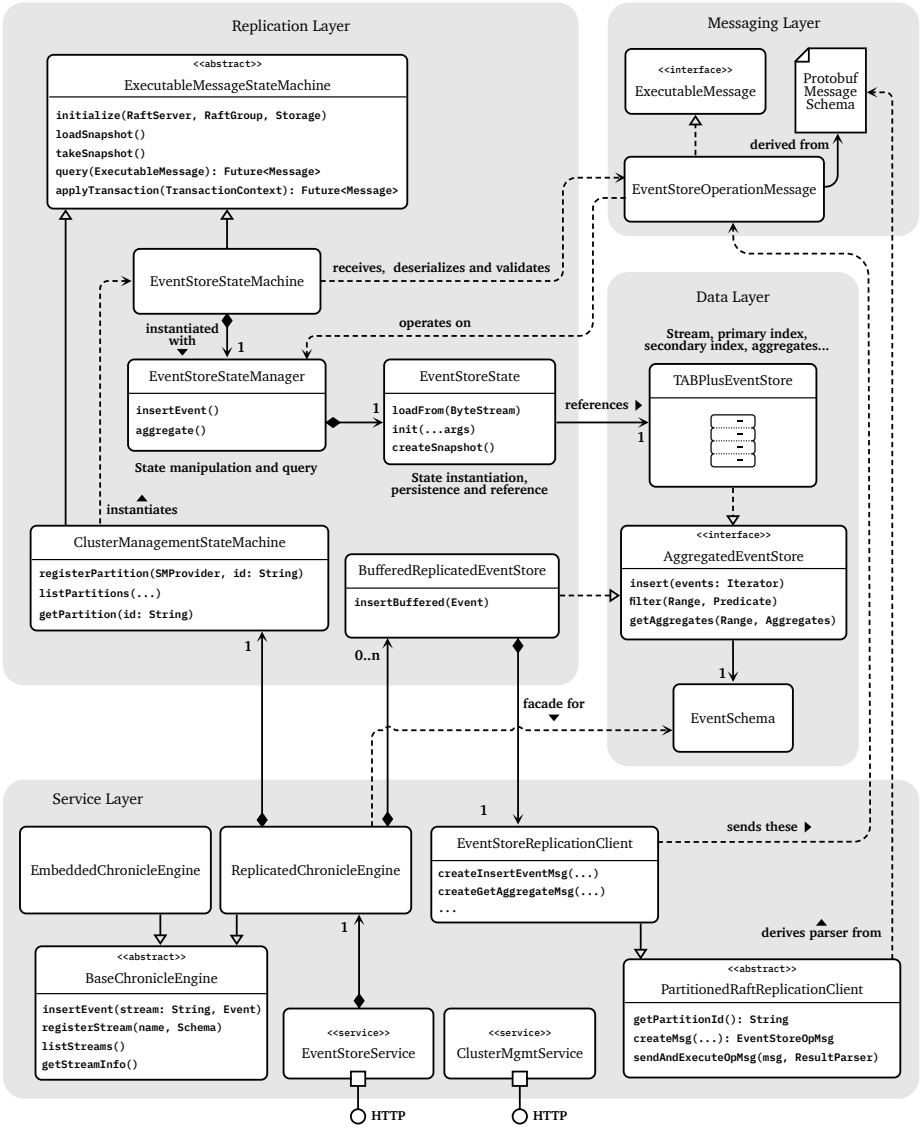
Out[21]:



Running the cluster without a buffer leads to 100% utilization of the machines IO, as in the current naive implementation of the raft log and state machine, each event is sent to all nodes, needs to be committed by at least a quorum of notes and is also written into the raft log of each node.

```
{localHostName=ip-172-26-0-78.eu-central-1.compute.internal, javaVersion=11,
localHostAddress=172.26.0.78, jdkVersion=11.0.14.1, storagePath=/home/ec2-user/chronicledb,
osName=Linux, springVersion=5.3.9, osVersion=4.14.262-200.489.amzn2.x86_64,
remoteHostAddress=3.121.183.166, nodeId=n1}
```

UML Class Diagrams



State

Persistent state on all servers:
(Updated on stable storage before responding to RPCs)

currentTerm	latest term server has seen (initialized to 0 on first boot, increases monotonically)
votedFor	candidateId that received vote in current term (or null if none)
log[]	log entries; each entry contains command for state machine, and term when entry was received by leader (first index is 1)

Volatile state on all servers:

commitIndex	index of highest log entry known to be committed (initialized to 0, increases monotonically)
lastApplied	index of highest log entry applied to state machine (initialized to 0, increases monotonically)

Volatile state on leaders:
(Reinitialized after election)

nextIndex[]	for each server, index of the next log entry to send to that server (initialized to leader last log index + 1)
matchIndex[]	for each server, index of highest log entry known to be replicated on server (initialized to 0, increases monotonically)

AppendEntries RPC

Invoked by leader to replicate log entries (§5.3); also used as heartbeat (§5.2).

Arguments:

term	leader's term
leaderId	so follower can redirect clients
prevLogIndex	index of log entry immediately preceding new ones
prevLogTerm	term of prevLogIndex entry
entries[]	log entries to store (empty for heartbeat; may send more than one for efficiency)
leaderCommit	leader's commitIndex

Results:

term	currentTerm, for leader to update itself
success	true if follower contained entry matching prevLogIndex and prevLogTerm

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. Reply false if log doesn't contain an entry at prevLogIndex whose term matches prevLogTerm (§5.3)
3. If an existing entry conflicts with a new one (same index but different terms), delete the existing entry and all that follow it (§5.3)
4. Append any new entries not already in the log
5. If leaderCommit > commitIndex, set commitIndex = min(leaderCommit, index of last new entry)

RequestVote RPC

Invoked by candidates to gather votes (§5.2).

Arguments:

term	candidate's term
candidateId	candidate requesting vote
lastLogIndex	index of candidate's last log entry (§5.4)
lastLogTerm	term of candidate's last log entry (§5.4)

Results:

term	currentTerm, for candidate to update itself
voteGranted	true means candidate received vote

Receiver implementation:

1. Reply false if term < currentTerm (§5.1)
2. If votedFor is null or candidateId, and candidate's log is at least as up-to-date as receiver's log, grant vote (§5.2, §5.4)

Rules for Servers

All Servers:

- If commitIndex > lastApplied: increment lastApplied, apply log[lastApplied] to state machine (§5.3)
- If RPC request or response contains term T > currentTerm: set currentTerm = T, convert to follower (§5.1)

Followers (§5.2):

- Respond to RPCs from candidates and leaders
- If election timeout elapses without receiving AppendEntries RPC from current leader or granting vote to candidate: convert to candidate

Candidates (§5.2):

- On conversion to candidate, start election:
  - Increment currentTerm
  - Vote for self
  - Reset election timer
  - Send RequestVote RPCs to all other servers
- If votes received from majority of servers: become leader
- If AppendEntries RPC received from new leader: convert to follower
- If election timeout elapses: start new election

Leaders:

- Upon election: send initial empty AppendEntries RPCs (heartbeat) to each server; repeat during idle periods to prevent election timeouts (§5.2)
- If command received from client: append entry to local log, respond after entry applied to state machine (§5.3)
- If last log index ≥ nextIndex for a follower: send AppendEntries RPC with log entries starting at nextIndex
  - If successful: update nextIndex and matchIndex for follower (§5.3)
  - If AppendEntries fails because of log inconsistency: decrement nextIndex and retry (§5.3)
- If there exists an N such that N > commitIndex, a majority of matchIndex[i] ≥ N, and log[N].term == currentTerm: set commitIndex = N (§5.3, §5.4).

A condensed summary of the Raft consensus algorithm (excluding membership changes and log compaction), extracted from the original Raft paper.

References

[1] A. Buchmann and B. Koldehofe, "Complex event processing," 2009, doi: /10.1524/itit.2009.9058.

[2] K. Cao, Y. Liu, G. Meng, and Q. Sun, "An overview on edge computing research," *IEEE access*, vol. 8, pp. 85714–85728, 2020, doi: 10.1109/ACCESS.2020.2991734.

[3] J. Wang, W. Zhang, Y. Shi, S. Duan, and J. Liu, "Industrial big data analytics: Challenges, methodologies, and applications," *arXiv preprint arXiv:1807.01016*, 2018, doi: 10.48550/arXiv.1807.01016.

[4] D. Mourtzis, E. Vlachou, and N. Milas, "Industrial big data as a result of IoT adoption in manufacturing," *Procedia cirp*, vol. 55, pp. 290–295, 2016, doi: 10.1016/j.procir.2016.07.038.

[5] M. Smuck, C. A. Odonkor, J. K. Wilt, N. Schmidt, and M. A. Swiernik, "The emerging clinical role of wearables: Factors for successful implementation in healthcare," *NPJ Digital Medicine*, vol. 4, no. 1, pp. 1–8, 2021, doi: 10.1038/s41746-021-00418-3.

[6] G. Liu, W. Zhu, C. Saunders, F. Gao, and Y. Yu, "Real-time complex event processing and analytics for smart grid," *Procedia Computer Science*, vol. 61, pp. 113–119, 2015, doi: 10.1016/j.procs.2015.09.169.

[7] J. R. Elias, R. Chard, M. Levental, Z. Liu, I. Foster, and S. Chaudhuri, "Real-time streaming and event-driven control of scientific experiments," *arXiv preprint arXiv:2205.01476*, 2022, doi: 10.48550/arXiv.2205.01476.

[8] A. Holzner and H. A. Council, "High throughput data acquisition at the CMS experiment at CERN." 2015.

[9] N. Hassan, K.-L. A. Yau, and C. Wu, "Edge computing in 5G: A review," *IEEE Access*, vol. 7, pp. 127276–127289, 2019, doi: 10.1109/ACCESS.2019.2938534.

[10] M. Seidemann, N. Glombiewski, M. Körber, and B. Seeger, "Chronicledb: A high-performance event store," *ACM Transactions on Database Systems (TODS)*, vol. 44, no. 4, pp. 1–45, 2019, doi: 10.1145/3342357.

[11] C. Konrad, "ChronicleDB on a raft." 2022, doi: 10.5281/zenodo.6991168.

[12] M. van Steen and A. S. Tanenbaum, *Distributed systems*. CreateSpace Independent Publishing Platform, 2007.

[13] Y. Liu and V. Vlassov, "Replication in distributed storage systems: State of the art, possible directions, and open issues," in *2013 international conference on cyber-enabled distributed computing and knowledge discovery*, 2013, pp. 225–232, doi: 10.1109/CyberC.2013.44.

[14] Kubernetes, "Kubernetes," *GitHub Repository*. https://github.com/kubernetes/kubernetes; GitHub, 2022.

[15] M. S. Nikoo, Ö. Babur, and M. Van Den Brand, "A survey on service composition languages," in *Proceedings of the 23rd ACM/IEEE international conference on model driven engineering languages and systems: Companion proceedings*, 2020, pp. 1–5, doi: 10.1145/3417990.3421402.

[16] Camunda, "Zeebe," *GitHub Repository*. https://github.com/camunda/zeebe; GitHub, 2022.

[17] R. Mkandla and E. Chikohora, "An evaluation of data consistency models in geo-replicated cloud storage," in *2021 3rd international multidisciplinary information technology and engineering conference (IMITEC)*, 2021, pp. 1–5.

[18] L. M. Silva, F. Aleixo, A. van der Linde, J. Leitão, and N. Preguiça, "Geo-located data for better dynamic replication," *arXiv preprint arXiv:2205.01045*, 2022, doi: 10.48550/arXiv.2205.01045.

[19] etcd.io, "Etcd," *GitHub Repository*. https://github.com/etcd-io/etcd; GitHub, 2015.

[20] C. Bi, S. Decker, K. Qian, and X. Yu, "CRaft DNS: A robust and scalable DNS server on raft," 2020.

[21] H. Khalajzadeh, D. Yuan, J. Grundy, and Y. Yang, "Cost-effective social network data placement and replication using graph-partitioning," in *2017 IEEE international conference on cognitive computing (ICCC)*, 2017, pp. 64–71, doi: 10.1109/IEEE.ICCC.2017.16.

[22] L. G. Williams and C. U. Smith, "Web application scalability: A model-based approach." in *Int. CMG conference*, 2004, pp. 215–226.

[23] S. M. Aaqib, "An efficient cluster-based approach for evaluating vertical and horizontal scalability of web servers using linear and non-linear workloads," in *2019 3rd international conference on trends in electronics and informatics (ICOEI)*, 2019, pp. 287–291, doi: 10.1109/ICOEI.2019.8862561.

[24] M. Mulazzani, "Reliability versus safety," *IFAC Proceedings Volumes*, vol. 18, no. 12, pp. 141–146, 1985, doi: 10.1016/S1474-6670(17)60097-1.

[25] S. U. Farooq, S. Quadri, and N. Ahmad, "Metrics, models and measurements in software reliability," in *2012 IEEE 10th international symposium on applied machine intelligence and informatics (SAMI)*, 2012, pp. 441–449, doi: 10.1109/SAMI.2012.6209008.

[26] J. D. Musa, "Operational profiles in software-reliability engineering," *IEEE software*, vol. 10, no. 2, pp. 14–32, 1993, doi: 10.1109/52.199724.

[27] Google, "Google 'protocol buffers'." https://web.archive.org/web/20220701130136/https://developers.google.com/protocol-buffers/docs/overview, 2022.

[28] A. Birolini, *Reliability engineering: Theory and practice*. Springer Science & Business Media, 2013.

[29] F. Cristian, "Understanding fault-tolerant distributed systems," *Communications of the ACM*, vol. 34, no. 2, pp. 56–78, 1991, doi: 10.1145/102792.102801.

[30] G. Candea and A. Fox, "Crash-only software," in *9th workshop on hot topics in operating systems (HotOS IX)*, 2003.

[31] G. Bracha and S. Toueg, "Resilient consensus protocols," in *Proceedings of the second annual ACM symposium on principles of distributed computing*, 1983, pp. 12–26, doi: 10.1145/800221.806706.

[32] R. D. Schlichting and F. B. Schneider, "Fail-stop processors: An approach to designing fault-tolerant computing systems," *ACM Transactions on Computer Systems (TOCS)*, vol. 1, no. 3, pp. 222–238, 1983, doi: 10.1145/357369.357371.

[33] R. van Renesse, Y. Minsky, and M. Hayden, "A gossip-style failure detection service," in *Middleware'98*, 1998, pp. 55–70.

[34] T. D. Chandra and S. Toueg, "Unreliable failure detectors for reliable distributed systems," *Journal of the ACM (JACM)*, vol. 43, no. 2, pp. 225–267, 1996, doi: 10.1145/226643.226647.

[35] A. Haeberlen, P. Kouznetsov, and P. Druschel, "The case for byzantine fault detection." in *HotDep*, 2006.

[36] H. Yamada and J. Nemoto, "Scalar DL: Scalable and practical byzantine fault detection for transactional database systems," *Proceedings of the VLDB Endowment*, vol. 15, no. 7, pp. 1324–1336, 2022, doi: 10.14778/3523210.3523212.

[37] R. H. Arpaci-Dusseau and A. C. Arpaci-Dusseau, "Fail-stutter fault tolerance," in *Proceedings eighth workshop on hot topics in operating systems*, 2001, pp. 33–38, doi: 10.1109/HOTOS.2001.990058.

[38] G. Kola, T. Kosar, and M. Livny, "Faults in large distributed systems and what we can do about them," in *European conference on parallel processing*, 2005, pp. 442–453, doi: 10.1007/11549468_51.

[39] A. Alquraan, H. Takruri, M. Alfatafta, and S. Al-Kiswany, "An analysis of {network-partitioning} failures in cloud systems," in *13th USENIX symposium on operating systems design and implementation (OSDI 18)*, 2018, pp. 51–68.

[40] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat, "Evolve or die: High-availability design principles drawn from googles network infrastructure," in *Proceedings of the 2016 ACM SIGCOMM conference*, 2016, pp. 58–72, doi: 10.1145/2934872.2934891.

[41] P. Gill, N. Jain, and N. Nagappan, "Understanding network failures in data centers: Measurement, analysis, and implications," in *Proceedings of the ACM SIGCOMM 2011 conference*, 2011, pp. 350–361, doi: 10.1145/2018436.2018477.

[42] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE transactions on dependable and secure computing*, vol. 1, no. 1, pp. 11–33, 2004, doi: 10.1109/TDSC.2004.2.

[43] M. Asplund, "Restoring consistency after network partitions," PhD thesis, Institutionen för datavetenskap, 2007.

[44] A. A. Helal, A. A. Heddaya, and B. B. Bhargava, *Replication techniques in distributed systems*, vol. 4. Springer Science & Business Media, 2006.

[45] A. W. Services, "Amazon S3 storage classes." https://web.archive.org/web/20201101005907/https://aws.amazon.com/s3/storage-classes/, 2020.

[46] A. W. Services, "Amazon S3 service level agreement." https://web.archive.org/web/20220517145126/https://aws.amazon.com/s3/sla/, 2022.

[47] Microsoft, "Microsoft CosmosDB service level agreement." https://web.archive.org/web/20210515125856/https://azure.microsoft.com/en-us/support/legal/sla/cosmos-db/v1_4/, 2021.

[48] D. Pritchett, "BASE: An acid alternative: In partitioned databases, trading some consistency for availability can lead to dramatic improvements in scalability." *Queue*, vol. 6, no. 3, pp. 48–55, 2008, doi: 10.1145/1394127.1394128.

[49] A. Fekete, *Consistency models for replicated data*, vol. 6. Springer, 2009.

[50] D. Bermbach, L. Zhao, and S. Sakr, "Towards comprehensive measurement of consistency guarantees for cloud-hosted data storage services," in *Technology conference on performance evaluation and benchmarking*, 2013, pp. 32–47, doi: 10.1007/978-3-319-04936-6_3.

[51] R. A. Campêlo, M. A. Casanova, D. O. Guedes, and A. H. Laender, "A brief survey on replica consistency in cloud environments," *Journal of Internet Services and Applications*, vol. 11, no. 1, pp. 1–13, 2020, doi: 10.1186/s13174-020-0122-y.

[52] H. Attiya and J. L. Welch, "Sequential consistency versus linearizability," *ACM Transactions on Computer Systems (TOCS)*, vol. 12, no. 2, pp. 91–122, 1994, doi: 10.1145/176575.176576.

[53] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," in *Communications of the ACM*, vol. 52, 1978, pp. 558–564.

[54] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 12, no. 3, pp. 463–492, 1990, doi: 10.1145/78969.78972.

[55] M. Shen, A. D. Kshemkalyani, and T. Hsu, "Causal consistency for geo-replicated cloud storage under partial replication," in *2015 IEEE international parallel and distributed processing symposium workshop*, 2015, pp. 509–518, doi: 10.1109/IPDPSW.2015.68.

[56] W. Lloyd, M. J. Freedman, M. Kaminsky, and D. G. Andersen, "Don't settle for eventual: Scalable causal consistency for wide-area storage with COPS," in *Proceedings of the twenty-third ACM symposium on operating systems principles*, 2011, pp. 401–416, doi: 10.1145/2043556.2043593.

[57] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, 2009.

[58] D. B. Terry, A. J. Demers, K. Petersen, M. J. Spreitzer, M. M. Theimer, and B. B. Welch, "Session guarantees for weakly consistent replicated data," in *Proceedings of 3rd international conference on parallel and distributed information systems*, 1994, pp. 140–149, doi: 10.1109/PDIS.1994.331722.

[59] E. A. Brewer, "Towards robust distributed systems," in *PODC*, 2000, vol. 7, pp. 343477–343502.

[60] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *Acm Sigact News*, vol. 33, no. 2, pp. 51–59, 2002, doi: 10.1.1.20.1495.

[61] E. Brewer, "CAP twelve years later: How the" rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, 2012, doi: 10.1109/MC.2012.37.

[62] M. Kleppmann, "A critique of the CAP theorem," *arXiv preprint arXiv:1509.05393*, 2015, doi: 10.48550/arXiv.1509.05393.

[63] D. Abadi, "Consistency tradeoffs in modern distributed database system design: CAP is only part of the story," *Computer*, vol. 45, no. 2, pp. 37–42, 2012, doi: 10.1109/MC.2012.33.

[64] J. M. Hellerstein and P. Alvaro, "Keeping CALM: When distributed consistency is easy," *Communications of the ACM*, vol. 63, no. 9, pp. 72–81, 2020, doi: 10.1145/3369736.

[65] S. Braun, "Keeping CALM - consistency in distributed systems made easy." https://www.slideshare.net/SusanneBraun2/keeping-calm-konsistenz-in-verteilten-systemen-leichtgemacht, 2022.

[66] S. Bagui and L. T. Nguyen, "Database sharding: To provide fault tolerance and scalability of big data on the cloud," *International Journal of Cloud Applications and Computing (IJCAC)*, vol. 5, no. 2, pp. 36–52, 2015.

[67] P. Kookarinrat and Y. Temtanapat, "Analysis of range-based key properties for sharded cluster of mongodb," in *2015 2nd international conference on information science and security (ICISS)*, 2015, pp. 1–4, doi: 10.1109/ICISSEC.2015.7370983.

[68] F. WEINGARTEN and J. LI, "Zero-downtime rebalancing and data migration of a mature multi-shard platform."

[69] J. Lamping and E. Veach, "A fast, minimal memory, consistent hash algorithm," *arXiv preprint arXiv:1406.2294*, 2014.

[70] P. Helland, "Standing on distributed shoulders of giants: Farsighted physicists of yore were danged smart!" *Queue*, vol. 14, no. 2, pp. 5–15, 2016, doi: 10.1145/2927299.2953944.

[71] J. Li, E. Michael, and D. R. Ports, "Eris: Coordination-free consistent transactions using in-network concurrency control," in *Proceedings of the 26th symposium on operating systems principles*, 2017, pp. 104–120, doi: 10.1145/3132747.3132751.

[72] U. Institute, "Annual outage analysis 2021." https://uptimeinstitute.com/annual-outage-analysis-2021; Uptime Institute, 2021.

[73] T.-Y. Hsu, A. D. Kshemkalyani, and M. Shen, "Causal consistency algorithms for partially replicated and fully replicated systems," *Future Generation Computer Systems*, vol. 86, pp. 1118–1133, 2018, doi: 10.1016/j.future.2017.04.044.

[74] A. Jeffery, H. Howard, and R. Mortier, "Rearchitecting kubernetes for the edge," in *Proceedings of the 4th international workshop on edge systems, analytics and networking*, 2021, pp. 7–12, doi: 10.1145/3434770.3459730.

[75] A. S. Dhillon, "An edge computing-based complex event processing technique for sensor-based systems," PhD thesis, Carleton University, 2018.

[76] G. Mondragón-Ruiz, A. Tenorio-Trigoso, M. Castillo-Cara, B. Caminero, and C. Carrión, "An experimental study of fog and cloud computing in CEP-based real-time IoT applications," *Journal of Cloud Computing*, vol. 10, no. 1, pp. 1–17, 2021, doi: 10.1186/s13677-021-00245-7.

[77] L. Lamport, "Specifying concurrent systems with TLA+," *Calculational System Design*, pp. 183–247, 1999, [Online]. Available: https://www.microsoft.com/en-us/research/publication/specifying-concurrent-systems-tla/.

[78] C. Newcombe, T. Rath, F. Zhang, B. Munteanu, M. Brooker, and M. Deardeuff, "Use of formal methods at amazon web services," 2014, [Online]. Available: https://6826.csail.mit.edu/2019/papers/formal-methods-amazon.pdf.

[79] A. Gotsman, H. Yang, C. Ferreira, M. Najafzadeh, and M. Shapiro, "'Cause i'm strong enough: Reasoning about consistency choices in distributed systems," in *Proceedings of the 43rd annual ACM SIGPLAN-SIGACT symposium on principles of programming languages*, 2016, pp. 371–384.

[80] M. Bravo, A. Gotsman, B. de Régil, and H. Wei, "UniStore: A fault-tolerant marriage of causal and strong consistency," in *2021 USENIX annual technical conference (USENIX ATC 21)*, 2021, pp. 923–937.

[81] A. W. Services, "Amazon S3 strong consistency." https://web.archive.org/web/20210509002449/https://aws.amazon.com/de/s3/consistency/, 2021.

[82] Microsoft, "Consistency levels in azure cosmos DB." https://web.archive.org/web/20220624103513/https://docs.microsoft.com/en-us/azure/cosmos-db/consistency-levels, 2022.

[83] P. Helland, "Immutability changes everything," *Communications of the ACM*, vol. 59, no. 1, pp. 64–70, 2015, doi: 10.1145/2844112.

[84] J. C. Corbett *et al.*, "Spanner: Google's globally distributed database," *ACM Transactions on Computer Systems (TOCS)*, vol. 31, no. 3, pp. 1–22, 2013, doi: 10.1145/2491245.

[85] E. Brewer, "Spanner, truetime and the cap theorem," 2017.

[86] R. H. Thomas, "A majority consensus approach to concurrency control for multiple copy databases," *ACM Transactions on Database Systems (TODS)*, vol. 4, no. 2, pp. 180–209, 1979, doi: 10.1145/320071.320076.

[87] L. Lamport, "Generalized consensus and paxos," 2005.

[88] D. Ongaro, "Consensus: Bridging theory and practice," PhD thesis, https://purl.stanford.edu/qr033xr6097; Stanford University, Stanford, CA, USA, 2014.

[89] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the ACM (JACM)*, vol. 32, no. 2, pp. 374–382, 1985, doi: 10.1145/3149.214121.

[90] P. Hunt, M. Konar, F. P. Junqueira, and B. Reed, "{ZooKeeper}: Wait-free coordination for internet-scale systems," in *2010 USENIX annual technical conference (USENIX ATC 10)*, 2010.

[91] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," in *Concurrency: The works of leslie lamport*, New York, NY, USA: Association for Computing Machinery, 2019, pp. 203–226.

[92] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM (JACM)*, vol. 27, no. 2, pp. 228–234, 1980, doi: 10.1145/322186.322188.

[93] Y. Xiao, N. Zhang, W. Lou, and Y. T. Hou, "A survey of distributed consensus protocols for blockchain networks," *IEEE Communications Surveys & Tutorials*, vol. 22, no. 2, pp. 1432–1465, 2020, doi: 10.1109/COMST.2020.2969706.

[94] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990, doi: 10.1145/98163.98167.

[95] J. Skrzypzcak and F. Schintke, "Towards log-less, fine-granular state machine replication," *Datenbank-Spektrum*, vol. 20, no. 3, pp. 231–241, 2020, doi: 10.1007/s13222-020-00358-4.

[96] K. Antoniadis, R. Guerraoui, D. Malkhi, and D.-A. Seredinschi, "State machine replication is more expensive than consensus," 2018. doi: 10.4230/LIPIcs.DISC.2018.7.

[97] M. Castro and B. Liskov, "Practical byzantine fault tolerance," in *Proceedings of the third symposium on operating systems design and implementation*, 1999, vol. 99, pp. 173–186, doi: 10.5555/296806.296824.

[98] T. Distler, "Byzantine fault-tolerant state-machine replication from a systems perspective," *ACM Computing Surveys (CSUR)*, vol. 54, no. 1, pp. 1–38, 2021.

[99] J. Gray, P. Helland, P. O'Neil, and D. Shasha, "The dangers of replication and a solution," in *Proceedings of the 1996 ACM SIGMOD international conference on management of data*, 1996, pp. 173–182, doi: 10.1145/233269.233330.

[100] M. Wiesmann and A. Schiper, "Comparison of database replication techniques based on total order broadcast," *IEEE Transactions on Knowledge and Data Engineering*, vol. 17, no. 4, pp. 551–566, 2005, doi: 10.1109/TKDE.2005.54.

[101] Y. Saito and M. Shapiro, "Optimistic replication," *ACM Computing Surveys (CSUR)*, vol. 37, no. 1, pp. 42–81, 2005.

[102] M. Letia, N. Preguiça, and M. Shapiro, "CRDTs: Consistency without concurrency control," *arXiv preprint arXiv:0907.0929*, 2009, doi: 10.48550/arXiv.0907.0929.

[103] K. De Porre, F. Myter, C. De Troyer, C. Scholliers, W. De Meuter, and E. G. Boix, "A generic replicated data type for strong eventual consistency," in *Proceedings of the 6th workshop on principles and practice of consistency for distributed data*, 2019, pp. 1–3, doi: 10.1145/3301419.3323974.

[104] S. Nair, F. Meirim, M. Pereira, C. Ferreira, and M. Shapiro, "A coordination-free, convergent, and safe replicated tree," *arXiv preprint arXiv:2103.04828*, 2021.

[105] Redis, "Redis - diving into conflict-free replicated data types (CRDTs)." https://web.archive.org/web/20220603195222/https://redis.com/blog/diving-into-crdts/, 2022.

[106] M. AdelsonVelskii and E. M. Landis, "An algorithm for the organization of information," JOINT PUBLICATIONS RESEARCH SERVICE WASHINGTON DC, 1963.

[107] N. Preguiça, J. M. Marquès, M. Shapiro, and M. Letia, "A commutative replicated data type for cooperative editing," in *2009 29th IEEE international conference on distributed computing systems*, 2009, pp. 395–403, doi: 10.1109/ICDCS.2009.20.

[108] D. Li and R. Li, "An admissibility-based operational transformation framework for collaborative editing systems," *Computer Supported Cooperative Work (CSCW)*, vol. 19, no. 1, pp. 1–43, 2010, doi: 10.1007/s10606-009-9103-1.

[109] Figma, "How figma's multiplayer technology works." https://web.archive.org/web/20220608054652/https://www.figma.com/blog/how-figmas-multiplayer-technology-works/, 2019, doi: 10.1145/1057977.1057980.

[110] J. Li, E. Michael, N. K. Sharma, A. Szekeres, and D. R. Ports, "Just say {NO} to paxos overhead: Replacing consensus with network ordering," in *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, 2016, pp. 467–483.

[111] R. Van Renesse and F. B. Schneider, "Chain replication for supporting high throughput and availability." in *OSDI*, 2004, vol. 4, doi: 10.5555/1251254.1251261.

[112] S. Almeida, J. Leitão, and L. Rodrigues, "ChainReaction: A causal+ consistent datastore based on chain replication," in *Proceedings of the 8th ACM european conference on computer systems*, 2013, pp. 85–98, doi: 10.1145/2465351.2465361.

[113] M. Burrows, "The chubby lock service for loosely-coupled distributed systems," in *Proceedings of the 7th symposium on operating systems design and implementation*, 2006, pp. 335–350.

[114] ACM, "Leslie lamport turing award." https://amturing.acm.org/award_winners/lamport_1205376.cfm, 2013.

[115] L. Lamport, "The part-time parliament," *ACM Transactions on Computer Systems 16, 2 (May 1998), 133-169. Also appeared as SRC Research Report 49. This paper was first submitted in 1990, setting a personal record for publication delay that has since been broken by [60].*, May 1998, doi: 10.1145/279227.279229.

[116] L. Lamport, "Paxos made simple," *ACM SIGACT News (Distributed Computing Column) 32, 4 (Whole Number 121, December 2001)*, pp. 51–58, 2001.

[117] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm (extended version)." Tech Report. May, 2014. http://ramcloud.stanford. edu/Raft. pdf, 2014, doi: 10.5555/2643634.2643666.

[118] L. Lamport, "Fast paxos," *Distributed Computing*, vol. 19, no. 2, pp. 79–103, 2006, doi: 10.1007/s00446-006-0005-x.

[119] J. Gray and L. Lamport, "Consensus on transaction commit," *ACM Transactions on Database Systems (TODS)*, vol. 31, no. 1, pp. 133–160, 2006, doi: 10.1145/1132863.1132867.

[120] S. Chand, Y. A. Liu, and S. D. Stoller, "Formal verification of multi-paxos for distributed consensus," in *International symposium on formal methods*, 2016, pp. 119–136, doi: 10.1007/978-3-319-48989-6_8.

[121] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *2011 IEEE/IFIP 41st international conference on dependable systems & networks (DSN)*, 2011, pp. 245–256, doi: 10.1109/DSN.2011.5958223.

[122] B. Oki and B. Liskov, "Viewstamped replication: A new primary copy method to support highly-available distributed systems," in *Proceedings of the seventh annual ACM symposium on principles of distributed computing*, 1988, pp. 8–17, doi: 10.1145/62546.62549.

[123] B. Liskov and J. Cowling, "Viewstamped replication revisited," 2012, doi: 1721.1/71763.

[124] X. Xu *et al.*, "A taxonomy of blockchain-based systems for architecture design," in *2017 IEEE international conference on software architecture (ICSA)*, 2017, pp. 243–252, doi: 10.1109/ICSA.2017.33.

[125] E. Androulaki *et al.*, "Hyperledger fabric: A distributed operating system for permissioned blockchains," in *Proceedings of the thirteenth EuroSys conference*, 2018, pp. 1–15, doi: 10.1145/3190508.3190538.

[126] H. Fabric, "Hyperledger fabric ordering service." https://web.archive.org/web/20220812064051/https://hyperledger-fabric.readthedocs.io/en/release-2.2/orderer/ordering_service.html, 2020.

[127] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, p. 21260, 2008.

[128] A. Wahab and W. Mehmood, "Survey of consensus protocols," *arXiv preprint arXiv:1810.03357*, 2018, doi: 10.48550/arXiv.1810.03357.

[129] N. Chaudhry and M. M. Yousaf, "Consensus algorithms in blockchain: Comparative analysis, challenges and opportunities," in *2018 12th international conference on open source systems and technologies (ICOSST)*, 2018, pp. 54–63, doi: 10.1109/ICOSST.2018.8632190.

[130] D. B. Kanga, M. Azzouazi, M. Y. El Ghoumrari, and A. Daif, "Management and monitoring of blockchain systems," *Procedia Computer Science*, vol. 177, pp. 605–612, 2020, doi: 10.1016/j.procs.2020.10.086.

[131] S. Zhou and S. Mu, "Fault-tolerant replication with pull-based consensus in MongoDB," in *18th USENIX symposium on networked systems design and implementation (NSDI 21)*, 2021, pp. 687–703, [Online]. Available: https://www.usenix.org/conference/nsdi21/presentation/zhou.

[132] C. Labs, "CockroachDB replication layer." https://web.archive.org/web/20220303101918/https://www.cockroachlabs.com/docs/stable/architecture/replication-layer.html, 2022.

[133] E. Sakic and W. Kellerer, "Response time and availability study of RAFT consensus in distributed SDN control plane," *IEEE Transactions on Network and Service Management*, vol. 15, no. 1, pp. 304–318, 2017, doi: 10.1109/TNSM.2017.2775061.

[134] R. Authors, "The secret lives of data." https://web.archive.org/web/20220714211518/https://thesecretlivesofdata.com/raft/, 2013.

[135] D. Ongaro, "Raft-dev mailing group: Raft vs. Viewstamped replication." https://web.archive.org/web/20220714211518/https://groups.google.com/g/raft-dev/c/cBNLTZT2q8o, 2015.

[136] H. Li, Z. Liu, and Y. Li, "An improved raft consensus algorithm based on asynchronous batch processing," in *INTERNATIONAL CONFERENCE ON WIRELESS COMMUNICATIONS, NETWORKING AND APPLICATIONS*, 2022, pp. 426–436, doi: 10.1007/978-981-19-2456-9_44.

[137] C. Copeland and H. Zhong, "Tangaroa: A byzantine fault tolerant raft." Tech. Rep, 2016.

[138] D. Tan, J. Hu, and J. Wang, "VBBFT-raft: An understandable blockchain consensus protocol with high performance," in *2019 IEEE 7th international conference on computer science and network technology (ICCSNT)*, 2019, pp. 111–115, doi: 10.1109/ICCSNT47585.2019.8962479.

[139] J. Clow and Z. Jiang, "A byzantine fault tolerant raft." Stanford University. Available at: http://www. scs. stanford. edu/17aucs244b . . . , 2017.

[140] etcd, "Etcd learner." https://web.archive.org/web/20220812064051/https://etcd.io/docs/v3.3/learning/learner/, 2021.

[141] V. Arora, T. Mittal, D. Agrawal, A. El Abbadi, X. Xue, *et al.*, "Leader or majority: Why have one when you can have both? Improving read scalability in raft-like consensus protocols," in *9th USENIX workshop on hot topics in cloud computing (HotCloud 17)*, 2017.

[142] Z. Xu, C. Stewart, and J. Huang, "Elastic, geo-distributed raft," in *2019 IEEE/ACM 27th international symposium on quality of service (IWQoS)*, 2019, pp. 1–9.

[143] C. Deyerl and T. Distler, "In search of a scalable raft-based replication architecture," in *Proceedings of the 6th workshop on principles and practice of consistency for distributed data*, 2019, pp. 1–7.

[144] Y. Wang, Z. Wang, Y. Chai, and X. Wang, "Rethink the linearizability constraints of raft for distributed key-value stores," in *2021 IEEE 37th international conference on data engineering (ICDE)*, 2021, pp. 1877–1882, doi: 10.1109/ICDE51399.2021.00170.

[145] C. Jensen and R. Snodgrass, *Temporal database*, vol. 6. Springer, 2009.

[146] A. Bolour, T. L. Anderson, L. J. Dekeyser, and H. K. Wong, "The role of time in information processing: A survey," *ACM SIGART Bulletin*, no. 80, pp. 28–46, 1982, doi: 10.1145/1056176.1056180.

[147] R. T. Snodgrass, "Temporal databases," in *Theories and methods of spatio-temporal reasoning in geographic space*, Springer, 1992, pp. 22–64.

[148] O. Etzion, *Event stream*, vol. 6. Springer, 2009.

[149] M. Körber, "Accelerating event stream processing in on-and offline systems," PhD thesis, Philipps-Universität Marburg, 2021.

[150] S. T. Watt, S. Achanta, H. Abubakari, E. Sagen, Z. Korkmaz, and H. Ahmed, "Understanding and applying precision time protocol," in *2015 saudi arabia smart grid (SASG)*, 2015, pp. 1–7, doi: 10.1109/SASG.2015.7449285.

[151] L. Liu and M. T. Özsu, *Encyclopedia of database systems*, vol. 6. Springer, 2009.

[152] Y. Ahmad, "Data stream management architectures and prototypes," in *Encyclopedia of database systems*, vol. 6, Springer, 2009.

[153] N. Glombiewski, P. Götze, M. Körber, A. Morgen, and B. Seeger, "Designing an event store for a modern three-layer storage hierarchy," *Datenbank-Spektrum*, vol. 20, no. 3, pp. 211–222, 2020, doi: 10.1007/s13222-020-00356-6.

[154] Inc. InfluxData, "Clustering in InfluxDB enterprise." https://web.archive.org/web/20220807023614/https://docs.influxdata.com/enterprise_influxdb/v1.9/concepts/clustering/, 2022.

[155] Inc. InfluxData, "InfluxDB and the raft consensus protocol." https://web.archive.org/web/20220807023614/https://s3.amazonaws.com/vallified/InfluxDBRaft.pdf, 2015.

[156] Inc. InfluxData, "InfluxDB design principles." https://web.archive.org/web/20220807023614/https://docs.influxdata.com/influxdb/cloud/reference/key-concepts/design-principles/, 2022.

[157] Inc. InfluxData, "Multiple data center replication with InfluxDB." https://web.archive.org/web/20220807023614/https://www.influxdata.com/blog/multiple-data-center-replication-influxdb/, 2017.

[158] C. Wang *et al.*, "Apache IoTDB: Time-series database for internet of things," *Proceedings of the VLDB Endowment*, vol. 13, no. 12, pp. 2901–2904, 2020, doi: 10.14778/3415478.3415504.

[159] Ltd. Event Store, "EventStoreDB—the database for event sourcing." https://web.archive.org/web/20220703090419/%0Ahttps://www.eventstore.com/eventstoredb, 2022.

[160] Ltd. Event Store, "EventStoreDB highly-available cluster." https://web.archive.org/web/20220703090419/%0Ahttps://developers.eventstore.com/server/v21.10/cluster.html#cluster-nodes, 2022.

[161] Inc. Coil Technologies, "TigerBeetle GitHub." https://web.archive.org/web/20220707023614/https://github.com/coilhq/tigerbeetle, 2022.

[162] Inc. Coil Technologies, "TigerBeetle." https://web.archive.org/web/20220707023614/https://www.tigerbeetle.com, 2022.

[163] QuestDB, "QuestDB homepage." https://web.archive.org/web/20220703090419/https://questdb.io/, 2022.

[164] QuestDB, "QuestDB benchmark suite." https://web.archive.org/web/20220703090419/https://questdb.io/time-series-benchmark-suite/, 2022.

[165] H.-A. Jacobsen, *Publish/subscribe*, vol. 6. Springer, 2009.

[166] A. Auradkar *et al.*, "Data infrastructure at LinkedIn," in *2012 IEEE 28th international conference on data engineering*, 2012, pp. 1370–1381, doi: 10.1109/ICDE.2012.147.

[167] O. Etzion, *Event channel*, vol. 6. Springer, 2009.

[168] G. Wang *et al.*, "Building a replicated logging system with apache kafka," *Proceedings of the VLDB Endowment*, vol. 8, no. 12, pp. 1654–1655, 2015, doi: 10.14778/2824032.2824063.

[169] A. Kafka, "KIP-101: Alter replication protocol to use leader epoch rather than high watermark for truncation." https://web.archive.org/web/20220316123624/https://cwiki.apache.org/confluence/display/KAFKA/KIP-101+-+Alter+Replication+Protocol+to+use+Leader+Epoch+rather+than+High+Watermark+for+Truncation#:~:text=The%20replication%20protocol%20in%20Kafka,will%20progress%20the%20High%20Watermark., 2022.

[170] A. Kafka, "KIP-500: Replace ZooKeeper with a self-managed metadata quorum." https://web.archive.org/web/20220316123624/https://cwiki.apache.org/confluence/display/KAFKA/KIP-500%3A+Replace+ZooKeeper+with+a+Self-Managed+Metadata+Quorum, 2022.

[171] RabbitMQ, "RabbitMQ quorum queues." https://web.archive.org/web/20210812064051/https://www.rabbitmq.com/quorum-queues.html, 2021.

[172] Microsoft, "Azure event hubs." https://web.archive.org/web/20220323043904/https://azure.microsoft.com/en-en/services/event-hubs/#features, 2022.

[173] Microsoft, "Availability and consistency in event hubs." https://web.archive.org/web/20220723051352/https://docs.microsoft.com/en-us/azure/event-hubs/event-hubs-availability-and-consistency, 2022.

[174] C. T. Yu and C. Chang, "Distributed query processing," *ACM computing surveys (CSUR)*, vol. 16, no. 4, pp. 399–433, 1984, doi: /10.1145/3872.3874.

[175] D. Kossmann, "The state of the art in distributed query processing," *ACM Computing Surveys (CSUR)*, vol. 32, no. 4, pp. 422–469, 2000, doi: 10.1145/371578.371598.

[176] E. Azhir, N. J. Navimipour, M. Hosseinzadeh, A. Sharifi, M. Unal, and A. Darwesh, "Join queries optimization in the distributed databases using a hybrid multi-objective algorithm," *Cluster Computing*, vol. 25, no. 3, pp. 2021–2036, 2022, doi: 10.1007/s10586-021-03451-9.

[177] R. S. Barga, J. Goldstein, M. Ali, and M. Hong, "Consistent streaming through time: A vision for event stream processing," *arXiv preprint cs/0612115*, 2006, doi: /10.48550/arXiv.cs/0612115.

[178] N. Marz, "How to beat the CAP theorem," *Thoughts from the Red Planet Blog*. https://web.archive.org/web/20220707023614/http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html, 2011.

[179] J. Lin, "The lambda and the kappa," *IEEE Internet Computing*, vol. 21, no. 5, pp. 60–66, 2017, doi: 1089-7801/17/$33.00.

[180] J. Li, K. Tufte, V. Shkapenyuk, V. Papadimos, T. Johnson, and D. Maier, "Out-of-order processing: A new architecture for high-performance stream systems," *Proceedings of the VLDB Endowment*, vol. 1, no. 1, pp. 274–288, 2008, doi: 10.14778/1453856.1453890.

[181] P. A. Tucker, D. Maier, T. Sheard, and L. Fegaras, "Exploiting punctuation semantics in continuous data streams," *IEEE Transactions on Knowledge and Data Engineering*, vol. 15, no. 3, pp. 555–568, 2003, doi: 10.1109/TKDE.2003.1198390.

[182] U. Srivastava and J. Widom, "Flexible time management in data stream systems," in *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on principles of database systems*, 2004, pp. 263–274, doi: 10.1.1.90.1199.

[183] G. Sharon, *Event causality*, vol. 6. Springer, 2009.

[184] J. Lindström and M. M. Elbushra, "Real-time eventual consistency," *International Journal in Foundations of Computer Science & Technology*, vol. 4, no. 4, pp. 63–76, 2014, doi: 10.5121/ijfcst.2014.4405.

[185] R. Authors, "Where can i get raft?" https://web.archive.org/web/20220714211518/https://raft.github.io/#implementations, 2022.

[186] A. S. Foundation, "Apache ratis." https://web.archive.org/web/20220331185632/https://ratis.apache.org/, 2022.

[187] A. S. Foundation, "Apache ratis source code," *GitHub Repository*. https://github.com/apache/ratis/tree/ratis-2.1.0; GitHub, 2022.

[188] A. S. Foundation, "Apache ozone." https://web.archive.org/web/20220328162102/https://ozone.apache.org/, 2022.

[189] HortonWorks, "High throughput data replication over RAFT." https://de.slideshare.net/Hadoop_Summit/high-throughput-data-replication-over-raft, 2018.

[190] S. Pedersen, H. Meling, and L. Jehl, "An analysis of quorum-based abstractions: A case study using gorums to implement raft," in *Proceedings of the 2018 workshop on advanced tools, programming languages, and PLatforms for implementing and evaluating algorithms for distributed systems*, 2018, pp. 29–35, doi: 10.1145/3231104.3231957.

[191] J. Iyengar, M. Thomson, *et al.*, "QUIC: A UDP-based multiplexed and secure transport," *Internet Engineering Task Force, RFC 9000*, 2021.

[192] M. Bishop, "HTTP/3," *Internet Engineering Task Force, RFC 9114*, 2022.

[193] L. Lawniczak and T. Distler, "Stream-based state-machine replication," in *2021 17th european dependable computing conference (EDCC)*, 2021, pp. 119–126, doi: 10.1109/EDCC53658.2021.00024.

[194] Z. Wang, C. Zhao, S. Mu, H. Chen, and J. Li, "On the parallels between paxos and raft, and how to port optimizations," in *Proceedings of the 2019 ACM symposium on principles of distributed computing*, 2019, pp. 445–454, doi: 10.1145/3293611.3331595.