TU WIEN Informatics

An Efficient Data Store for a Dependable Distributed Control Unit

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Daniel Achleitner, BSc
Matrikelnummer 00926807

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Dipl.-Ing. Dr.techn. Thomas Frühwirth, BSc
Mitwirkung: Dipl.-Ing. Thomas Preindl, BSc
Dipl.-Ing. Stefan Seifried, BSc

Wien, 14. Oktober 2024

Daniel Achleitner   Thomas Frühwirth

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Informatics

An Efficient Data Store for a Dependable Distributed Control Unit

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Daniel Achleitner, BSc
Registration Number 00926807

to the Faculty of Informatics

at the TU Wien

Advisor: Dipl.-Ing. Dr.techn. Thomas Frühwirth, BSc
Assistance: Dipl.-Ing. Thomas Preindl, BSc
Dipl.-Ing. Stefan Seifried, BSc

Vienna, October 14, 2024

Daniel Achleitner   Thomas Frühwirth

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Erklärung zur Verfassung der Arbeit

Daniel Achleitner, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 14. Oktober 2024

Daniel Achleitner

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Acknowledgements

I would like to thank Wolfgang Kastner and Edgar Eidenberger for the opportunity to work on this topic at the intersection of two fascinating areas of computer science. I would also like to thank Thomas Preindl and Stefan Seifried for all their efforts, patience and the consistent persistence—thanks for taking me on a journey that led to a considerable expansion of my knowledge and involved many sessions of productive discussion and interesting ideas. And finally, my deepest gratitude to my parents for their support and understanding over the years.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Kurzfassung

Brandmeldeanlagen erfüllen eine sicherheitskritische Funktion und müssen zuverlässig arbeiten. Daher wird eine dezentrale Architektur verwendet, wenn größere Bereiche oder mehrere Gebäude abgedeckt werden sollen: Autonome Steuereinheiten werden zu einer verteilten Steuereinheit vernetzt und teilen sich einen gemeinsamen logischen Zustand. Um Fehlertoleranz und hohe Zuverlässigkeit zu gewährleisten, kann ein kontinuierlich replizierender verteilter Datenspeicher verwendet werden. Wie die CAP- und PACELC-Theoreme zeigen, müssen dabei jedoch Kompromisse eingegangen werden zwischen Datenkonsistenz, Verfügbarkeit und Latenzzeit.
Im Zuge dieser Arbeit untersuchen wir, ob schwache Konsistenzmodelle – die bessere Verfügbarkeit, Latenzzeit und Nebenläufigkeit ermöglichen können – für einen korrekten und sicheren Betrieb auf Hardware mit limitierten Ressourcen ausreichen. Wir identifizieren das schwächste geeignete Konsistenzmodell und entwickeln eine passende Architektur für einen Datenspeicher, der dieses Modell auf eingebetteten Geräten bereitstellen kann. Zunächst werden aus den Produktkriterien der europäischen Normenreihe EN 54 die Anforderungen gesammelt und verwandte Arbeiten zu Konsistenzmodellen und Replikation in verteilten Systemen gesichtet. Dann entwickeln wir, dem Forschungsansatz der Design Science folgend, iterativ eine Architektur und parallel dazu einen Prototyp. Der Prototyp ermöglicht es, Probleme des Designs frühzeitig zu erkennen. Schließlich bewerten wir das Design qualitativ anhand der Anforderungen und testen den Prototyp mit einer Simulationssoftware. Diese Software kann Netzwerkprobleme wie Latenzzeiten und Verluste simulieren, um deren Auswirkungen auf den Betrieb zu messen. Das aufgezeichnete Protokoll kann zur Überprüfung der Korrektheit in Bezug auf das beabsichtigte Konsistenzmodell verwendet werden.
Wir zeigen, dass das FIFO/PRAM-Konsistenzmodell den Anforderungen genügt. Das Modell stellt sicher, dass alle Teilnehmer die Schreiboperationen eines Teilnehmers in genau der Reihenfolge verarbeiten, in der sie von diesem Teilnehmer durchgeführt wurden. Wir schlagen eine Erweiterung des NDC-Frameworks vor und bauen unser Design darauf auf. Das Design beinhaltet einen Algorithmus, der FIFO-konsistente Sichten sicherstellt und das Risiko von Prioritätsumkehr ausschließt. Optimierungen für das Anti-Entropie-Protokoll tragen dazu bei, die Latenzzeit gering zu halten. Die Auswertung zeigt, dass das Design die Anforderungen erfüllen kann. Die Testergebnisse zeigen, dass der Prototyp in der Lage ist, eine konsistente Sicht bei Netzwerkfehlern aufrechtzuerhalten, allerdings auf Kosten einer leicht erhöhten Latenzzeit im Vergleich zu Eventual Consistency.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Abstract

Fire Detection and Fire Alarm systems have a safety-critical role and must operate reliably. Therefore, a decentralized architecture is used when large areas or multiple buildings need to be covered: Autonomous control units are networked together to form a distributed control unit and share common logical state. To ensure fault tolerance and high reliability, a continuously replicating distributed data store may be used. However, as shown by the CAP and PACELC theorems, there are trade-offs involved between data consistency, availability, and latency.

In this work, we investigate whether relaxed consistency models—which can obtain better availability, latency and concurrency—can be sufficient for correct and safe operation on resource constrained hardware. We identify the weakest consistency model suitable for the purpose, and develop a software architecture for a data store that can provide the model on constrained devices.

First, we gather requirements from the product criteria defined by the European Standards series EN 54 and conduct a review of existing work on consistency models and replication in distributed systems. Then, following a design science approach, we iteratively develop a software architecture in parallel to a prototype. The prototype allows early identification of problems with the design. Finally, we evaluate the design qualitatively against the requirements and test the prototype with simulation testing software. The software can introduce networking artifacts—such as latency and message loss—to measure their effects on the executed operations. The recorded history can be used to verify correctness with respect to the intended consistency model.

We show that the FIFO/PRAM consistency model is adequate. It ensures that all nodes observe write operations from one node in the order they were issued by the node. We propose an extension of the node-wide dot-based clocks (NDC) framework and base our design on it. The design incorporates an algorithm that provides FIFO consistent views while ruling out the risk of priority inversion scenarios. Optimizations for the anti-entropy protocol help to keep latency low. The evaluation finds that the design is able to meet the requirements. The testing results show that the implemented prototype is able to maintain a consistent view under network faults, at the cost of a slightly increased visibility latency compared to eventual consistency.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Contents

Kurzfassung ix

Abstract xi

Contents xiii

1 Introduction 1
- 1.1 Problem Statement and Motivation . . . 1
- 1.2 Aim of the Thesis and Expected Results . . . 2
- 1.3 Methodology . . . 3

2 Background 5
- 2.1 Fire Detection and Fire Alarm Systems . . . 5
- 2.2 Distributed Systems . . . 14
- 2.3 Consistency Models . . . 22

3 Related Work 31
- 3.1 Consistency Models . . . 31
- 3.2 Network and Safety Architecture . . . 36
- 3.3 Consensus and Replication . . . 40

4 Requirements 49
- 4.1 Existing Solution . . . 49
- 4.2 Requirements Analysis . . . 54

5 System Design 63
- 5.1 System Model . . . 63
- 5.2 Dot-Based Clocks Framework . . . 67
- 5.3 Consistency Models . . . 69
- 5.4 Delivery Tracking . . . 82
- 5.5 Fault Detection . . . 83
- 5.6 Storage Durability and Crash Recovery . . . 84
- 5.7 Large Value Objects . . . 87
- 5.8 Networking Layer Improvements . . . 88

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

6 Evaluation **91**
6.1 Qualitative Analysis . . . 91
6.2 Testing . . . 97

7 Conclusion **105**

A NDC Framework Extensions **107**
A.1 Priority-Aware FIFO Algorithm . . . 107

Overview of Generative AI Tools Used **113**

List of Figures **115**

List of Tables **117**

Acronyms **119**

Bibliography **123**

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 1

Introduction

1.1 Problem Statement and Motivation

Fire Detection and Fire Alarm Systems (FDASs) must operate reliably and be able to tolerate faults to fulfill their safety-critical role. Modern systems often need to cover multiple areas, floors, or buildings. For this reason, they employ a decentralized architecture: Detector and alarm devices such as smoke detectors, manual call points, sounders, or visual alarm devices, are attached to control units using an electrical bus. Multiple autonomous control units, in turn, are connected to each other via redundant network links to form a Distributed Control Unit (DCU) that can cover a wider area. The control units can share a common logical state by using a distributed data store (DDS). To be able to provide fault tolerance and high availability, such a data store has to replicate data continuously.

Distributed systems, however, are hard to design—there are a number of technical trade-offs involved between data consistency, availability and latency. Every real network is unreliable and can introduce undesirable artifacts. Messages can be delivered late, out-of-order, duplicated multiple times, or get lost entirely. These artifacts can cause nodes of a shared-data system to observe different views of the same logical shared data. They make reasoning about and testing of a distributed application difficult. For this reason, data store systems often provide consistency models. A consistency model is a guarantee a data store can provide to a client application, promising that it will shield the application from the effects of the underlying unreliable network and only show consistent views.

Strong consistency models, such as Linearizability or Sequential Consistency, ensure that all nodes share a very similar view, but they have drawbacks: For example, such a data store might have to pause any processing of data in case some nodes are not reachable via the network—to prevent the occurrence of inconsistencies such as stale data or lost

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

writes. According to the CAP theorem, in case of a network partition, a shared-data system can either provide strict consistency or availability, but not both. Furthermore, the PACELC theorem extends this by the notion that even in normal operation, the system has to choose between preferring either strict consistency or lower latency. These theorems indicate that while weaker consistency models may require an application to be able to tolerate more inconsistencies, they can enable better tolerance of network partitions, better availability and lower latency for data operations.

We observe that not all control functions of an FDAS share the same demands for reliability and latency: The most critical functions, i.e., alarming in direct proximity to a fire or smoke hazard, typically do not depend on any data from other control units. They can be handled autonomously within the control unit itself. For other functions, modest delays may be acceptable: Safety functions operating in greater physical distance to a hazard, such as in a different fire protection zone or in an adjacent building, generally have more relaxed requirements. In less critical functions, it may even suffice to reliably detect and report a malfunction of the system. Functionality related to maintenance, such as sensor testing or the distribution of firmware and configuration updates, typically has service personnel already present on site. In that situation, performance may be more important than safety so that the system is fully operational again more quickly. The different functional safety demands on the distributed data store need to be considered separately for resource constrained devices.

In this work, we focus on a distributed data store for the purpose of an FDAS and the required consistency model. Due to the mentioned fundamental trade-offs, weak consistency models are able to obtain better availability in the event of a network partition. They also have advantages when the network is healthy: Communication latency, resource usage, concurrency, and throughput can improve. These benefits might enable additional use cases and simplify engineering.

1.2 Aim of the Thesis and Expected Results

We want to examine whether relaxed consistency models can be sufficient for correct and safe operation on resource constrained hardware.
Towards this aim, two research questions are defined:

RQ–1: Suitable Consistency Model

What is the weakest consistency model that is suitable for the purpose of a Fire Detection and Fire Alarm System?

RQ–2: Suitable Technology and Architecture

What technologies and which software architecture can be used to provide that consistency model on constrained devices?

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

1.3 Methodology

1.3.1 Literature Review

Starting out, we review the existing work on distributed systems. This includes consistency models, guidelines for network and safety architectures and algorithms for consensus and replication in distributed data stores.

1.3.2 Requirements Engineering

To be able to determine a suitable consistency model, we have to gather the requirements for an FDAS application first. We briefly analyze an existing system for its properties and limitations. We review the European Standards series EN 54, which provides relevant product standards and application guidelines. Based on these sources, we can identify functional and non-functional requirements for a data store design. We are then able to answer Research Question RQ–1.

1.3.3 System Design

Based on the gathered requirements and the determined consistency model, we follow a design science approach [Hev+04] to iteratively develop and evaluate a proposal for a system design. Building a prototype in parallel allows to identify any problems with the design early. Because we are analyzing the problem in the context of resource constrained devices, we take care to select technologies appropriate for that context. We then can compare the required properties of the found model with the properties of the found candidates, choose and combine them and iteratively refine the result. Based on the proposed system design, we can answer Research Question RQ–2.

1.3.4 Evaluation

We qualitatively evaluate the proposed system design created in the previous step against the functional and non-functional requirements. Then, we use specialized testing software to analyze the behavior of our prototype implementation. We utilize the Maelstrom workbench [Kinb], which includes the Jepsen testing library [KPa]. Maelstrom can simulate a number of clients that run randomly generated data operations against the data store. Maelstrom records the invocation and completion or failure of every operation in a history, which can then be analyzed for compliance to the defined consistency model. In addition, a graphical timeline of the operations and statistical data is generated, allowing to draw conclusions about the correctness of the implementation. Various network artifacts of unreliability can be introduced into the simulated system to study the effects on the generated history and its correctness. This includes conditions such as network partitions, latency and message loss.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Structure of the Work

The remainder of the work is structured as follows: The next Chapter 2 introduces important concepts and terminology related to FDASs, distributed systems and consistency models. Chapter 3 continues with a discussion of existing work related to consistency models, network and safety architecture, as well as consensus and replication in distributed data stores. Chapter 4 presents our gathered requirements for a suitable data store and answers RQ–1. The following Chapter 5 presents our proposed system design discusses the weighed trade-offs and answers RQ–2. In Chapter 6, we qualitatively evaluate the design against the gathered requirements. We also use simulation testing software for distributed data stores to measure our prototype and verify that it can provide the required consistency model. Finally, we discuss our results and findings in Chapter 7.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 2

Background

The background chapter introduces concepts and terminology which is required for the rest of the work. This includes basic aspects of Fire Detection and Fire Alarm Systems (FDASs), the trade-offs involved in distributed systems (DSs) and the well-known consistency models a distributed data store can offer to simplify the development of client applications.

2.1 Fire Detection and Fire Alarm Systems

Fire safety is a paramount concern in buildings and industrial facilities. Fires can lead to severe consequences, including loss of life, property damage and economic setbacks due to production outages. Apart from preventative measures such as integrating fire safety policies in building codes, early detection and intervention is essential.

Automated FDASs perform a critical role in fire safety. They continuously monitor the environment for signs of smoke or fire and can take early intervention action automatically. The building occupants are warned through audible and visual alarm signals. Additional action can include alerting emergency responders to the site and communicating with building control systems to minimize the spread and to facilitate evacuation efforts.

2.1.1 Components and Terminology

To help define the components and terminology, we refer to the relevant European standards' series EN 54, "Fire detection and fire alarm systems". The EN 54 standards defines product characteristics, test methods and performance criteria for evaluating and certifying FDAS components [EN 21]. The separate parts concern themselves with different topics and product groups, as listed in Table 2.1. As a minimum useful configuration, a system typically consists of detector and alarm devices, a control panel, a power supply and the necessary cabling between these components.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Part	Group	Title
EN 54–1	Introduction	Introduction
EN 54–2	Control panels and power supply	Control and indicating equipment
EN 54–3	Alarm devices	Sounders
EN 54–4	Control panels and power supply	Power supply equipment
EN 54–5	Detectors	Point heat detectors
EN 54–7	Detectors	Point smoke detectors
EN 54–10	Detectors	Flame detectors
EN 54–11	Detectors	Manual call point
EN 54–12	Detectors	Line smoke detectors
EN 54–13	System compatibility	Compatibility and connectability assessment of system components
EN 54–23	Alarm devices	Visual alarm devices
EN 54–24	Alarm devices	Loudspeakers

Table 2.1: EN 54 standard series (excerpt) [EN 21]

Detectors

Detectors (initiating devices) can be manual or automatic. Manual call points or pull stations are installed in easily reachable locations and can be actuated by any occupant that observes smoke or fire [EN 05b]. Automatic fire detectors are sensors that monitor their environment using different physical measurement principles. When the measured values cross a predetermined pre-alarm or alarm threshold, the control unit is notified. These sensors include smoke detectors which sense the presence of smoke particles in the air [EN 18b], heat detectors which detect rapid rise in temperature [EN 18a] or flame detectors which detect open flames by monitoring specific optical wavelengths [EN 05a].

Alarm Devices

Alarm devices (notification devices) alert building occupants to the possible danger to facilitate a timely evacuation. These actors include audible alarm devices (sounders) [EN 19b], which emit a distinctive sound, and loudspeakers which play back voice instructions for evacuation [EN 08]. Visual alarm devices, which emit bright, flashing lights, are used to help to alert occupants with hearing impairments [EN 10].

Control Panels and Power Supply

The central and most essential component is the Control and Indicating Equipment (CIE), Fire Alarm Control Panel (FACP) or Fire Alarm Control Unit (FACU). It monitors detector signals and activates notification devices when necessary. The control panel displays the system status and allows control for service and emergency personnel. Records of any notable events are kept for later inspection. To power the whole system, a reliable power supply is required. In the event of a power outage, it must keep delivering

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

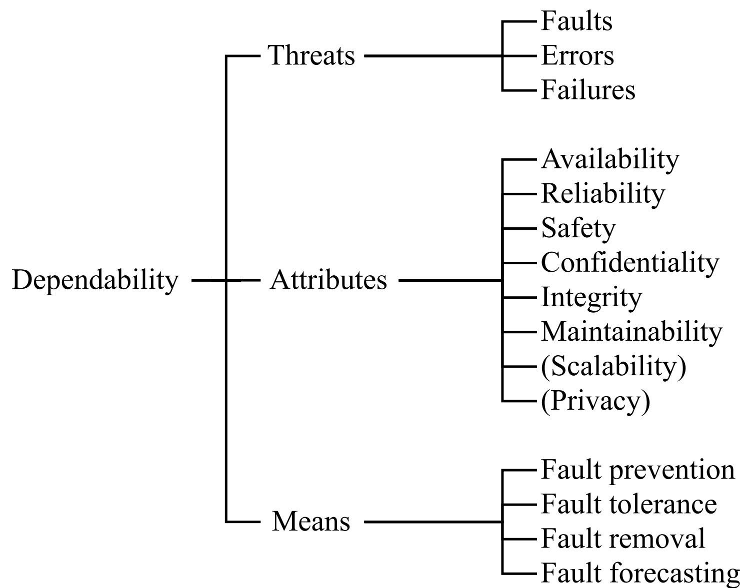


Figure 2.1: The dependability tree [ALR01; FKK15]

power from a secondary source for some guaranteed duration, e.g., by using batteries or an emergency generator [EN 06].

Communication with External Systems

FDASs often interface with external systems. They can be connected to other building control systems to minimize the spread of dangerous smoke fumes and the fire itself. This typically involves disabling elevator controls, limiting air movement, improving ventilation or closing fire safety doors. Outside the building, they can be connected to the fire department or other external monitoring services for fast emergency response and to ensure timely maintenance in case of system malfunctions.

2.1.2 Dependability

Dependability of a computing system is the ability to deliver service that can justifiably be trusted. The concepts of dependability can be separated into three main parts: the threats to, the attributes of and the means by which dependability can be achieved, as shown in the taxonomy in Figure 2.1 [ALR01].

Threats

The root cause of a malfunction is called a *fault*. A fault can be attributed to the influence of physical effects, or it can be human-made, intentionally or unintentionally. It can either lay dormant or it can be transiently, intermittently or permanently activated and produce an error. An *error* is the part of the system state that may lead to a subsequent

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

failure. It is an internal, hidden discrepancy between the intended and the actual state. If measures are taken to detect and correct the error, it may never surface as a failure. A *failure* happens when the error reaches the system boundary and causes the delivered service to visibly deviate from the intended behavior, failing to meet its specification. Such a deviation in a subsystem, in turn, can become the fault to the supersystem it belongs to, where it might again cause another error. This causal relationship between faults, errors, and failures is called the *fundamental chain of threats* (Figure 2.2) [ALR01; FKK15].

Figure 2.2: The fundamental chain of threats to dependability [ALR01]

Attributes

The dependability attributes are the properties that are expected of a dependable computing system. *Availability* is the readiness for usage of a service (uptime), while *reliability* is the probability that a system will conform to its specification after a given time. These two properties can be defined in a quantitative manner. *Safety* is the absence of catastrophic consequences on the user and the environment. *Security* entails the "CIA triad", the summation of the three properties of *availability* (for authorized users), *confidentiality* (keeping data private) and *integrity* (keeping data intact). *Maintainability* is the system's capability to be repaired and modified. *Scalability* and *privacy* are not present in the classic dependability tree of Avizienis, Laprie, and Randell, but they are important for Internet of Things (IoT) devices and embedded computing [FKK15].

Means

The means for dependable systems are the techniques that are used to provide these properties. *Fault prevention* includes procedures that are followed to prevent faults during the design, development and manufacturing phases, e.g., adhering to guidance, following standards and performing quality control. *Fault tolerance* aims to put mechanisms in place to continue delivery of the specified service in the presence of active faults which could not be prevented. It is commonly implemented by employing redundancy. *Fault removal* requires detecting and correcting faults, both during the development phase and the operational life of the system. During development, it consists of three steps: verification (testing), diagnosis and correction. Once a system is put into operation, it consists of corrective or preventative maintenance. *Fault forecasting* is done by evaluating the system behavior qualitatively and quantitatively to predict and address likely faults [ALR01].

2.1.3 Networking and Distribution of Control Units

Reliability is an important consideration in FDASs. Standards and local regulation specify requirements for reliability to ensure that the system functions as intended during

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

emergencies. These include allowed false alarm rates, mandated detection capabilities and limits to response times.

Redundancy

Generally, all hardware and software components are imperfect. To achieve high reliability despite being composed of such flawed components, a system must be tolerant to faults—it must be able to continue operating as a whole despite some of its components having failed. To do that, it must be designed with built-in *redundancy*: extra components that can take over the function of failed components. Redundancy can be *in space* (e.g., building a system with more hardware components than needed) or *in time* (e.g., performing the same operation or sending the same data multiple times) [Sto12].

Centralized Systems

In smaller buildings, a single and centralized FDAS can be sufficiently reliable as critical key components implemented redundantly. Power supplies can fall back from mains to battery power, a stand-by secondary processing unit can take over from the primary unit. The wiring inside the building includes alternate, redundant transmission paths. Often, a ring topology is used to connect detectors and alarm devices, so that one single wiring fault such as a short or an open can be detected and tolerated. Centralized units have the advantage that installation and configuration of the system can be relatively simple and easy to reason about.

Decentralized Systems

In larger buildings or expansive facilities like an industrial plant or a commercial airport, however, using only a single control unit would pose significant limitations and risks for engineering, maintenance and the resulting reliability:

Limited Scalability A single control unit has limitations regarding the number of zones and devices it can manage, both from the electrical interfacing and its logical processing capabilities. Raising these limits by engineering a "larger" control unit may be an expensive endeavor that is not viable.

Single Point of Failure A single control unit represents a single point of failure. If it malfunctions or becomes compromised, the problem may lead to a complete failure of the system. Such an impact can be economically severe by interrupting normal business and production operations.

Maintenance and Testing Performing maintenance and testing on a large, centralized control unit can be logistically challenging. Shutting down all fire protection for a larger facility at once is typically not acceptable.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

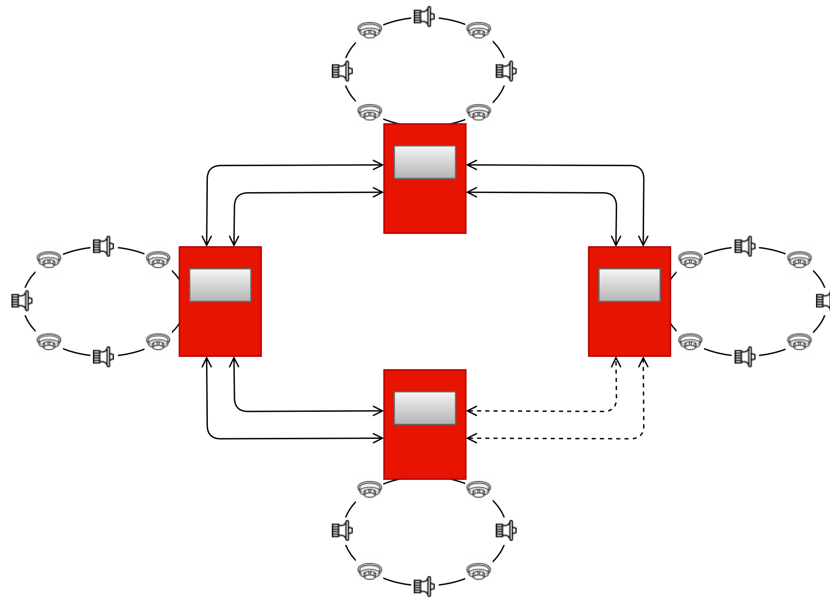


Figure 2.3: Example topology of a networked FDAS; consisting of four control units, with redundant connections and local rings of detector and alarm devices[1]

To avoid these problems, multiple control units are installed for large facilities. They still protect their assigned areas autonomously, but they can be networked to form a virtual Distributed Control Unit (DCU) together (Figure 2.3). This allows to implement control logic that acts across different control units. To some extent, the involved control units present and act as one single virtual distributed system. By partially hiding the networking complexity, existing planning and configuration know-how from single-unit systems can still be used. Maintenance tasks such as monitoring for faults, testing detector functionality or updating configuration and firmware can be performed more efficiently, too.

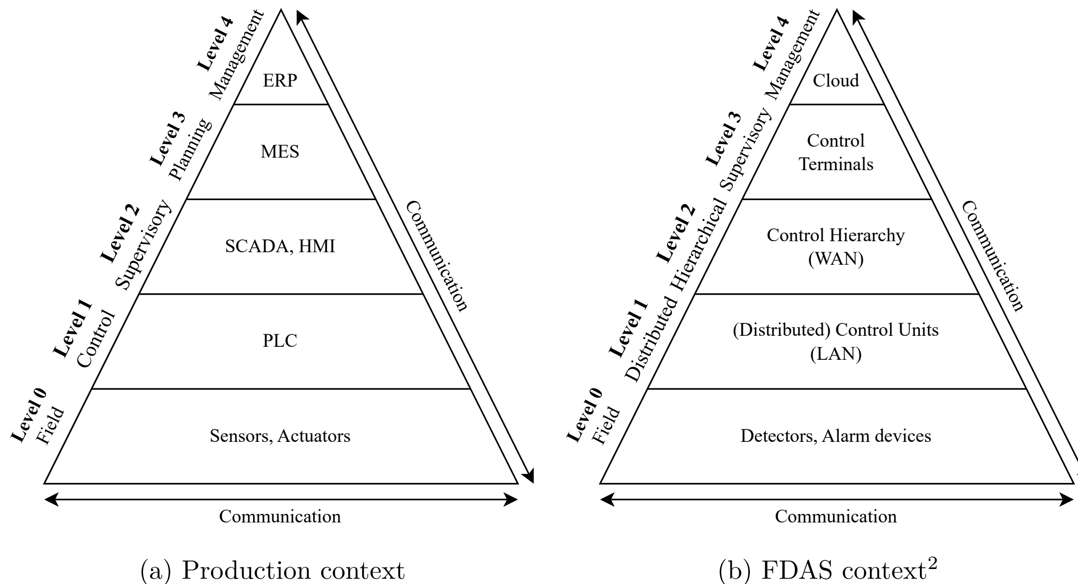
Distributed systems, however, are generally hard to design—there are a number of technical trade-offs involved, which we discuss in the following Section 2.2. Safety-relevant functions must continue to work reliably and consistently according to their specifications, no matter from which unit data originates from and on which unit it is processed.

Networked CIEs in EN 54

EN 54–01 clarifies the use of terms within the series of standards [Annex B EN 21, p. 20]: A *distributed* Control and Indicating Equipment (CIE) in the EN 54 is a *single control unit* that is distributed into multiple physical cabinets, connected by some transmission path. At least one specific cabinet provides manual controls and indications (a control

[1] Icons: flaticon.com

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



(a) Production context

(b) FDAS context[2]

Figure 2.4: The Automation Pyramid

panel), but each of the cabinets can have functional devices attached, in particular detector and alarm devices. A *network* of CIEs in the EN 54, in turn, is a network of multiple CIEs, where each node can be either standalone or distributed as defined above. When we refer to distributed systems or DCUs in this work, we always mean the latter.

EN 54–13, *Fire detection and fire alarm systems—Part 13: Compatibility and connectability assessment of system components* [EN 19a] specifies requirements a system using networked CIEs must adhere to. Most importantly, transmission time for important conditions must not exceed 20 seconds, transmission paths must be fault-tolerant and a faulty unit must be detected by at least one other unit. We consider these requirements in Chapter 4.

2.1.4 Automation Pyramid

In industrial settings, the overall architecture for automation systems is often illustrated using the automation pyramid. It is grounded in the standards of ISA-95 and IEC 62264, but many variations exist in the literature. Typically, the pyramid depicts the functional hierarchy within a manufacturing enterprise, as shown in Figure 2.4a. Communication occurs in two different dimensions: horizontally between peers within the same layer and vertically between systems on different layers.

At the top, corporate IT systems are responsible for high-level business planning, logistics and managing manufacturing operations (ERP, MES). This happens over regular IT

[2]Adaption: Stefan Seifried

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

networks and does not impose particularly high demands for reliability and the timeframes involved, which can range from hours to months. In the middle, at level 2, monitoring and supervision systems (SCADA, HMI) bridge the gap from the IT world into the operational technology (OT) world of industrial control systems (ICSs). Towards the bottom, we get into the detailed field level. Sensors provide data for programmable logic controllers (PLCs), which control the actuators to actually run the physical production process. Here, specialized field networks or buses are in use, which typically provide high assurances for reliability, availability, and integrity. Timeframes for control loops are in the seconds to microseconds range [KF09; Nie+18].

In Figure 2.4b, we have adapted the pyramid to our purpose of a FDAS. In this case, there is no need for a planning level. However, we have split the control level into two distinct levels with different concerns.

Level 1 (Distributed) The lower control level contains the individual control units which serve roughly the same role as PLCs: communicating with field devices within its assigned area, e.g., a specific building, processing detector data and triggering alarm devices. These can work autonomously to ensure the safety relevant functionality for the area, regardless of the state of other parts of the system. They can be networked together to form distributed control units, sharing their data store.

Level 2 (Hierarchical) The upper control level forms a hierarchy of control units of a broad area, aggregating the data of the lower levels. This could include multiple buildings belonging to an industrial site, an airport or even multiple locations throughout a city or country. Because of the more limited bandwidth and higher latency, the aggregated data is only selectively forwarded as far as required.

To ensure the safety relevant functionality cannot be compromised, any communication flowing downwards from an upper level must be safeguarded. The installed firmware and critical configuration parameters have to be protected from unauthorized or unintended changes. Care must also be taken that the functionality cannot be impaired by accidental or malicious communication errors. If the upper levels simply keep to passive monitoring, the communication from any higher level can simply be rejected.

2.1.5 Resource Constraints

The small, embedded computers that implement the function of a FDAS are constrained devices. They need to operate under severely limited resources such as limited processing power and small amounts of available volatile and non-volatile memory.

In RFC 7228, Bormann, Ersue, and Keränen define constrained nodes and networks:

Constrained Nodes A *constrained node* is a resource constrained device communicating over and being part of a network. Some of the characteristics that are otherwise

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

taken for granted are not attainable on a constrained node. This sets it apart from the more typical and powerful computing nodes on the Internet like server systems, desktop/laptop computers or modern smartphones. Optimization for energy and bandwidth usage is a dominating consideration for the design requirements. Typical constraints include limited code complexity (ROM and Flash storage), size of state and buffers (RAM), constraints on the amount of computation achievable per period of time (processing power) and power usage. Additionally, when deployed in the field, user interfaces and accessibility may be limited, e.g., to modify the configuration or deploy updated software [BEK14].

Constrained Networks A *constrained network* is a network where some characteristics taken for granted in common link layers are not attainable. Typical constraints include low achievable bitrate or throughput, high packet loss, high variability of packet loss, severe penalties for using larger packets or limits on reachability over time [BEK14].

Constrained-Node Networks A *constrained-node network* is a network that is being primarily composed of constrained nodes. It is always a constrained network because its characteristics are influenced by the constraints of the individual nodes [BEK14].

While distributed systems research often makes an effort to measure and optimize for network constraints such as bandwidth usage, throughput and latency, constraints on the nodes themselves are rarely considered. This context has to be kept in mind for researching related work (Chapter 3), as it restricts the usefulness of many well-established algorithms and technologies.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2.2 Distributed Systems

Tanenbaum and Steen provide the following definition as a loose characterization of a distributed computing system: “A distributed system is a collection of independent computers that appears to its users as a single coherent system.” ([TS06])

To combine the two aspects of the definition and to achieve common goals as one system, these independent computers have to collaborate and interact through some interconnection to share data and coordinate activities. In general, the geographical scope of a distributed system can vary greatly. It might consist of just a few integrated circuits that communicate over a shared electrical bus, or it might consist of a planet-wide network of nodes communicating over the Internet.

2.2.1 Goals

There are different reasons for building a computer system in a distributed fashion. Achieving these goals make the high effort and complexity involved in development and operation worthwhile:

Resource Sharing Making it easy for users and applications to access remote resources in a controlled and efficient way.

Distribution Transparency Hiding the fact that the processes and resources are physically distributed across multiple computers. This includes access transparency, location transparency and concurrency transparency.

Openness Offering services according to standard rules, providing portability and interoperability.

Scalability Allowing for a system to grow easily with respect to some dimension, e.g., its size of users or resources, its geographical dimensions or its manageability.

Fault Tolerance Enabling a system as a whole to stay available to some degree despite malfunctions of one or more components.

2.2.2 Challenges

Designing distributed systems that meet these goals while hiding the complexity and staying easy to use is a difficult task. There are many technical challenges and trade-offs involved where there is no single optimal solution that fits all possible application requirements. Distributed systems differ from traditional software in that the components must communicate over a network. Any real-world network, however, is unreliable and has inherent undesirable properties whose effects must be taken into account and planned for at design time of the system.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Fallacies of Distributed Computing

Peter Deutsch and others at Sun Microsystems famously formulated the "Fallacies of Distributed Computing", the false assumptions which programmers new to distributed systems tend to make [TS06]:

1. The network is reliable.
2. The network is secure.
3. The network is homogenous.
4. The topology does not change.
5. Latency is zero.
6. Bandwidth is infinite.
7. Transport cost is zero.
8. There is one administrator.

These fallacies reveal all the properties that must be taken into account when designing a distributed system: A network is often unreliable, insecure, heterogeneous, etc. What makes these properties problematic from an engineering standpoint is that they typically get worse with increasing size and complexity of the system. Development and testing is typically done with small and reliable local networks where the caused problems can easily stay latent and hidden. But they can lead to catastrophic results much later once the system reaches a certain size or complexity, or an unexpected fault happens. Once triggered, these problems are also hard to debug: Their nondeterministic nature can make it hard to reproduce the effects reliably and to monitor the activity of the affected components and connections.

Unreliable Networks

Data packets or messages sent via unreliable, asynchronous networks such as the Internet are subject to networking conditions that can introduce various artifacts. They may get dropped and not arrive at the receiver at all. Or, if they indeed do arrive, they may be received with an arbitrary delay, duplicated multiple times, or in a different order than they were sent in (Figure 2.5). For an endpoint, it is not possible to tell whether a missing message is merely delayed or if it has been dropped for good—a dropped message is a special case of an infinitely delayed message. A network condition that prevents messages between two nodes from being delivered is called a *network partition* [Kle17].

Unreliable Clocks

Many embedded devices do not include real-time clocks. Instead, they often keep track of elapsed time using relatively inaccurate timestamp counter registers. Since these start counting when a device was powered on or was last reset, the absolute timestamps of different devices cannot be compared. Even for devices that do include real-time clocks,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

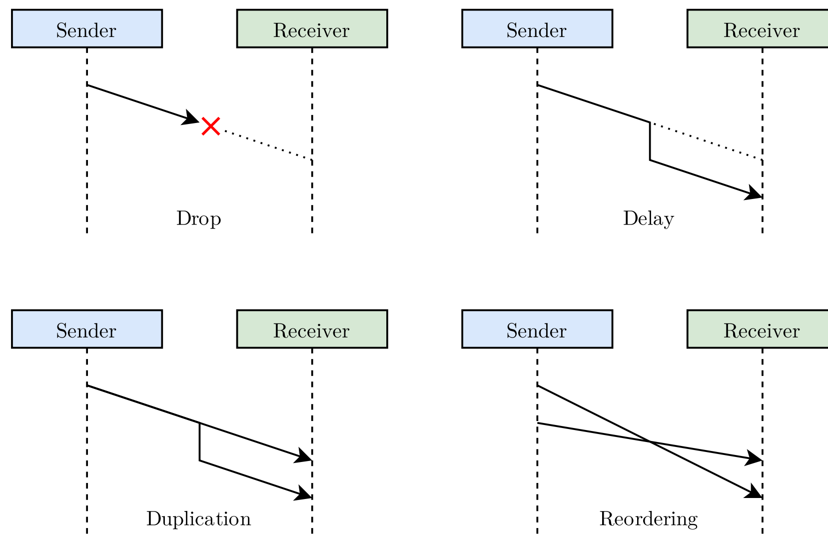


Figure 2.5: Communication artifacts in unreliable, asynchronous networks

they are of limited use to solve the synchronization problems in a distributed system: Even if synchronized periodically, they typically do not run accurately enough. To an application running on the device, the clock may appear to jump back or forth when time is adjusted during synchronization with an external source such as a Network Time Protocol (NTP) server. Other common reasons for adjustment include Daylight Saving Time (DST) coming into effect or leap seconds being applied to account for irregularities in Earth's rotation. These can cause wildly inaccurate results when calculating a duration by simply subtracting two captured timestamps.

To avoid discontinuity problems, many platforms provide *monotonic clocks*. Monotonic clocks guarantee to yield continuously increasing timestamps and can be useful for distributed systems, but working with them has its own pitfalls. The difference between two monotonic timestamp values can be used to calculate the elapsed duration of time, but the absolute value is meaningless. It is only meaningful on the associated system, but cannot be used with timestamps gathered from different systems.

Instead, distributed systems often employ *logical clocks*. Logical clocks do not measure elapsed time, but consist of logical counters within the distributed system. They are incremented on distributed events and can be used to safely and consistently establish a relative ordering. *Lamport timestamps* use a simple algorithm to provide a partial ordering of events. *Vector clocks* are an extension of the idea which can be used to infer possible causality between non-concurrent events [Lam78; Kle17].

Unreliable Components

In addition to the network being unreliable in transporting messages between the computers that make up a distributed system, the computers themselves are not reliable.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

A lot of things can go wrong, from hardware problems to software bugs, leading to complete or partial faults. It is also possible that correctly working components interact in unfortunate ways to break some design assumption of the distributed system. For instance, a very slow I/O-operation, a garbage collection run, or a suspended virtual machine might pause an important process long enough for the other nodes to detect it as unreachable or faulted. They then have to take appropriate measures, e.g., choosing a new replication leader. When the paused process eventually continues, it is important that instead of simply continuing with its pending tasks, it detects and adapts to the new situation. In case of a change in leadership, it has to recognize the new leader and synchronize its state accordingly [Kle17].

2.2.3 Distributed Data Store

A distributed data store (DDS) or shared-data system is a distributed system where shared data is stored on more than one node. One reason is to keep latency low by keeping data geographically close to the clients. Another common reason is to increase availability and fault tolerance by keeping the system working even when some individual parts have failed, e.g., some nodes being faulty or the network connections between some nodes being interrupted. Data is also often shared to improve scalability, to increase throughput and balancing the load by distributing queries onto multiple machines [Kle17].

Clients can connect to a node and invoke operations to query or manipulate the state of a shared data object, e.g., to read, write, increment or compare-and-swap (CAS) a value. An operation takes some time to complete and can either succeed or fail. Typically, clients can access the shared data with full location transparency, i.e., without any concern for where and how it is actually stored and updated.

Replication

Any changes to the data, however, must be propagated to other nodes so that their attached clients can see the most recent state. This process is called replication and every copy of the data is called a replica. Because different clients can concurrently invoke operations on the same data objects and the underlying unreliable network can drop, delay, duplicate, or reorder messages unpredictably, arbitrary inconsistency problems can happen: Conflicting operations may result in replicas applying different writes or writes getting lost. Clients may not receive the most recent state on a read. A read operation can even result in an older state being returned than on an earlier read, virtually "going backwards in time" [Kle17].

These non-deterministic artifacts can make reasoning about and testing of a distributed application very difficult. Subtle implementation flaws may lay dormant for a long time, but can result in disastrous bugs and faults if one day, an "unlucky" timing or fault situation occurs. A system design can make efforts to prevent these artifacts from happening even when the underlying network is unreliable: Such a contract is called a *consistency model* (see Section 2.3).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Leaders and Followers

There are three different approaches to implement replication, each coming with different advantages and disadvantages. They differ in conflict resolution and the avoidance of stale data:

Single-leader replication Clients send all writes to a designated *leader* node, which applies the operation and informs all other nodes (the *followers*) of the changes. For reads, they can query any node, but may receive stale data due to replication lag. This approach is popular and easy to reason about: since all writes go through the leader node, data conflicts do not arise. Properly recovering from faulty nodes and network interruptions, however, has some complexity where a lot can go wrong: a recovering follower node has to query and apply all missing changes. A faulty or unreachable leader node requires a *failover*: the promotion of a follower node to a new leader and all clients being reconfigured to send writes to the new leader. A recovering leader node has to detect whether a failover occurred, and, if so, demote itself to a follower node and apply all missing changes [Kle17].

Multi-leader replication Clients send all writes to one of several *leader* nodes, which can apply the operation and inform the other leaders and any *follower* nodes of the changes. This approach can be more robust and provide better performance, but it is harder to implement and to reason about. Due to the possibility of concurrent writes, conflicts and consistency problems can occur and have to be prevented or detected and resolved [Kle17].

Leaderless replication Clients send all writes to several nodes in parallel. For reads, they also have to query several nodes in parallel to detect stale data and bring it up to date. Similar to multi-leader replication, this approach can be robust in the presence of faulty nodes and network interruptions at the cost of higher complexity. Since there are no leaders and followers, clients have to participate in the process of resolving conflicts and consistency problems [Kle17].

Partitioning

Instead of keeping multiple copies of the same data on multiple nodes, a dataset can also be distributed by breaking it up into multiple partitions or shards that are stored on different nodes. The main reason to do that is to improve scalability, so that datasets that have grown too big to fit onto one node can be stored. Partitioning can also help to distribute the query load evenly, avoiding hot spot nodes with a disproportionately high load. However, the partitioning scheme to use has to be chosen very carefully, depending on the query requirements of the application and expected the dataset structure. Partitioning is usually combined with replication, so that even though each data object belongs to exactly one partition, it may still be stored on several different nodes for fault tolerance [Kle17].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Quorums

There are many situations and distributed algorithms where the computers comprising a distributed system must come to an unanimous decision, e.g., to choose a new leader node in single-leader system or to decide on accepting a read or a write operation in a leaderless system. The decision cannot exclusively rely on a single node, because the node may fail at any time, leaving the system stuck. One common solution is to rely on a *quorum*: every node may cast a vote and a successful decision requires a minimum number of votes. Often, a *majority quorum* is used which requires votes from strictly more than half the total nodes in the cluster ($\frac{N}{2} + 1$).

An advantage of the method is that a few failed or unreachable nodes do not prevent a decision and the system can keep working. How many unreachable nodes can be tolerated can be set by selecting the quorum condition accordingly. Since there can only be one majority in the cluster, decisions cannot conflict and the result is safe to use to avoid inconsistency. A drawback is that voting can increase latency, which impacts performance, and that in case of too many failed or unreachable nodes, operations cannot continue, which impacts availability. Another drawback is that the number of total nodes needs to be known to define a majority and to choose an appropriate quorum condition. If nodes are allowed to join and leave freely, the current number can be difficult to assess [Kle17].

2.2.4 CAP Theorem: Consistency vs. Availability

The CAP theorem or Brewer's conjecture was presented by Eric Brewer in 2000 [Bre00] and formally proven by Gilbert and Lynch [GL02]. It describes a trade-off that states that any shared-data system can fulfill at most two of the following three properties:

Consistency Every read receives either the most recent write or an error.

Availability Every read or write request receives a non-error response.

Partition Tolerance The system continues to operate as a whole despite some messages being delayed or dropped between nodes.

In principle, the design of a system can forfeit any one of these properties to ensure that the remaining two can be upheld (shown in Figure 2.6). Under real-world conditions, however, network partitions do invariably happen sooner or later, so every distributed data store must be able to tolerate them. Until such a partition is resolved, and the data can be reconciled, an implementation must either continue running operations to stay available (AP), or refrain from processing any operations at all to avoid inconsistency (CP). The first risks inconsistency due to potentially stale data, while the latter reduces availability. Hence, the theorem can be more clearly phrased as: If there is a network partition, a data store has to choose between preferring consistency or availability [Bre00; GL02].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2.2.5 PACELC Theorem: Consistency vs. Latency

The PACELC theorem was published by Abadi in 2012. It tries to address the problem that the CAP theorem falls short on describing the desirable properties and trade-offs involved in the design of a distributed data store. Abadi suspects that system architects who wanted to gain high availability simply followed the CAP theorem and reduced the consistency guarantees of their system. This conclusion is flawed: While network partitions do invariably happen, they are usually a somewhat rare occurrence. In their absence, the CAP theorem does not impose any restrictions on the possible consistency/availability guarantees. It is possible for a system to uphold both properties simultaneously.

During the normal, non-partitioned operation, however, another fundamental trade-off exists that deserves the same kind of academic scrutiny: A data store has to choose between either preferring better consistency or lower latency. The reason is that a data store has to replicate data continuously in advance to be able to provide availability in case of a network partition. The replication can happen synchronously or asynchronously:

- Synchronous replication requires waiting for confirmation from all replicas before acknowledging success to the writer. This ensures *consistency*, but increases the latency of the write operation.
- Asynchronous replication, in contrast, can acknowledge a write operation immediately without waiting for any confirmations. This keeps the *latency* of the write operation low. Because inconsistent reads can occur on the other replicas in the meantime, however, consistency is reduced. Alternatively, if a read request is routed to a specific master node and served from there, *consistency* can be ensured, but latency of the read operation may increase instead.
- Hybrid combinations of these two modes, where only some subset of replicas is updated synchronously, show the same latency/consistency trade-off depending on the chosen protocol.

To account for this trade-off, Abadi introduced PACELC [Aba12] as an extension to the CAP theorem. Its letters serve as a useful system to categorize the behavior of distributed computer systems which replicate data (shown in Figure 2.7): In the case of a network partition (P), how does the system trade off availability (A) and consistency (C)? This case corresponds to the CAP theorem. Or else (E), when the system is running normally in the absence of any network partitions, how does the system trade off latency (L) and consistency (C)?

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

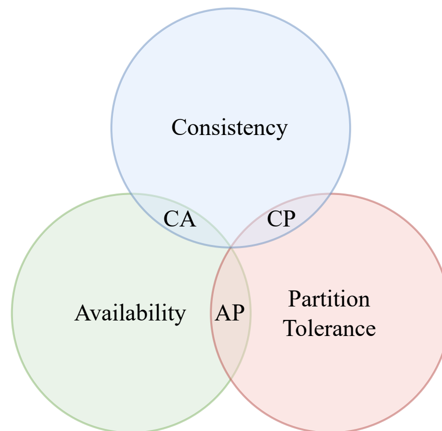


Figure 2.6: Venn diagram illustrating the CAP theorem

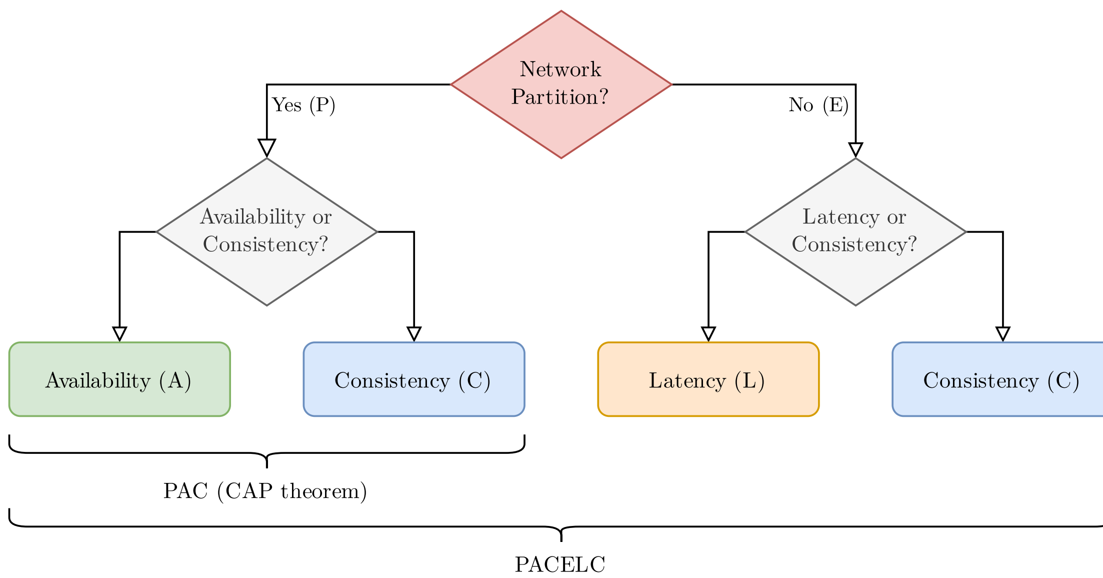


Figure 2.7: Flowchart illustrating the PACELC decision tree

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

2.3 Consistency Models

In their proof of CAP, Gilbert and Lynch used the definition of atomic or linearizable consistency:

> "There must exist a total order on all operations such that each operation looks as if it were completed at a single instant. This is equivalent to requiring requests of the distributed shared memory to act as if they were executing on a single node, responding to operations one at a time." ([GL02; Aba12])

This traditional notion of consistency is the strongest and strictest consistency model for non-transactional, single-object systems: *linearizability*. While many earlier system designs aimed to adhere to this strong model, modern systems often implement weaker and more permissive consistency models to provide better availability, lower latency and better scalability instead. The popularity of the CAP theorem itself is partially responsible for this shift [Aba12].

As mentioned in Section 2.2.2, replicating distributed data stores suffer from undesirable artifacts which can make reasoning and testing hard for system designers and application developers. A consistency model is a contract an underlying data store can provide to the application using them. It can state that it prevents these artifacts from happening at all or, at least, from the application being exposed to them. There is a basic trade-off involved as these benefits do not come for free. Stronger or stricter consistency models are difficult to implement in a data store, but they can make the data store easy to use for the client application. They restrict the attainable level of availability and scalability and may incur higher latency. Weaker or more permissive consistency models, in contrast, can enable higher availability, lower latency and scalability, at the cost of the data store being significantly harder to use for the client application.

2.3.1 Relationships between Consistency Models

Figure 2.8 illustrates the relationship hierarchy of common consistency models. A directed edge from model A to model B indicates that B implies A, i.e., that any execution that satisfies model B also satisfies model A. An execution or history is a collection of operations, including the relative order of their invocation and completion. The models in the upper part of the diagram have stricter or stronger consistency semantics, while the models in the lower part have more permissive or weaker semantics.

The models in the left half of the diagram are transactional models, where operations (transactions) can act on multiple objects at once. These models are similar to, but not equal, to the traditional isolation levels as defined in the literature for database systems [Bai+13b]. The models in the right half of the diagram are non-transactional models, where the available read and/or write operations can only act on single values or objects. The distributed systems' literature typically concerns itself predominantly with these kinds of operations [VV15].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

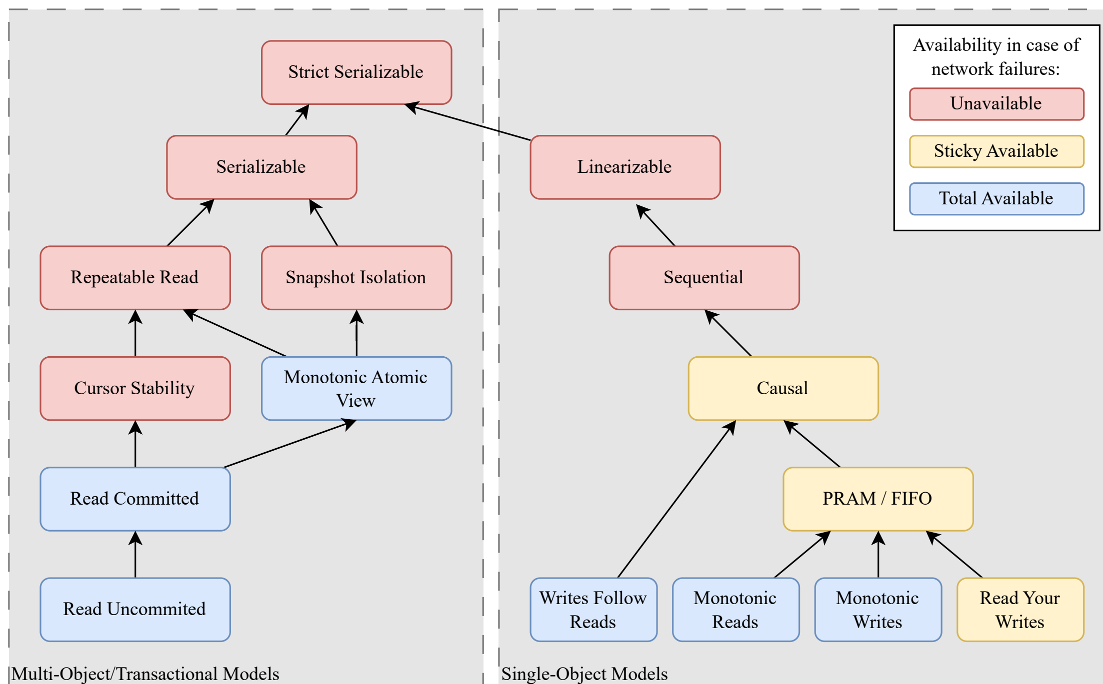


Figure 2.8: Relationship hierarchy between consistency models [Kina; Bai+13b; VV15]

Availability

Not all models in Figure 2.8 can provide availability and keep processing operations in case of a network partition or failure. The node coloring indicates the kind of availability that is possible, i.e., that a careful implementation can provide:

Unavailable Data stores guaranteeing any of these strong models can only be available as long as the network is not faulty—otherwise, some or all nodes have to stop processing operations to maintain the high expectations of consistency.

Sticky Available Data stores guaranteeing any of these weaker models are available on non-faulty nodes, as long as connected clients do not switch between different logical replicas across subsequent operations.

Total Available Data stores guaranteeing any of these weak models are available on non-faulty nodes, even when the network is completely down and when clients do switch between different logical replicas. The weak consistency expectations can be maintained regardless of the network conditions.

There are two different ways for clients to achieve *sticky availability*: by caching their writes and keeping a local copy of data, clients can trivially maintain stickiness by becoming a node themselves. However, such a cache can grow quite large and involved,

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

since in case of partial replication it might have to contain data of multiple servers. The second way is to use sticky routing on the network level, so that clients' requests are always routed to the same server. This method can also be difficult: There needs to be some way to identify the same client on subsequent requests and a way to deal with a previously contacted server not being available anymore due to faults or maintenance [Bai+13b; Baib].

2.3.2 Eventual Consistency

Eventual consistency, also called optimistic replication, is a very weak consistency model. It merely ensures that, if no new updates are made to some data item, eventually all replicas will converge to a common state and all reads will return the most recent value.

Eventual consistency relies on two properties: *total propagation* requires that any write operation eventually reaches each server, so that no replica can stay at an old state indefinitely. Additionally, *consistent ordering* requires that all servers do apply (non-commutative) write operations in the same order, so that all replicas eventually converge to the same state. The model does not impose any limit on how far a replica may fall behind—it might be hardly noticeable, only fractions of a second, or very significant intervals such as hours or days. This drawback can result in high implementation complexity for the distributed application, but the trade-off is often worthwhile since the data store can provide high performance and total availability. Stronger consistency models trivially include eventual consistency, otherwise they would not be very useful [Ter+94; Vog08; BD13].

Anti-Entropy

Due to unreliable networks or nodes, write operations can always get delayed or lost during replication. To guarantee the *total propagation* property, eventually consistent data stores typically employ anti-entropy protocols. This secondary mechanism keeps the data of all replicas in sync in case the primary delivery mechanism has failed. In other words, anti-entropy is used "to lower the convergence time bounds in eventually consistent systems" [Pet19, p. 244]. These mechanisms can either run on read (*read repair*), periodically as a background process (*anti-entropy process*), or as some combination of both. They compare the contents of different replicas to find missing updates. These are then applied to reconcile the contained records and "repair" the contents of the data store [Kle17, p. 178].

Conflict-Free Replicated Data Types

Conflict-free replicated data types (CRDTs) are data structures useful to implement eventual consistency. Initially defined by Letia, Preguiça, and Shapiro in 2010 [LPS10], they can ensure consistency for mutable data in the large scale at a low cost and are an active and promising area of research. They are categorized into two different types: state-based CRDTs and operation-based CRDTs.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

For a state-based CRDT, the merge operation is responsible for merging any received state updates with the own local state. The operation must satisfy three properties so that all replicas arrive at the same result deterministically and without conflicts:

- *Commutativity*: application order does not matter ($A \lor B = B \lor A$)
- *Associativity*: application preference does not matter ($(A \lor B) \lor C = A \lor (B \lor C)$)
- *Idempotence*: multiple applications do not matter ($A \lor A = A$)

Then, any replica can simply be changed locally without external coordination or any other complex concurrency control—all replicas will converge to the same common state as soon as the same collected write operations from all replicas are applied. These properties make CRDTs particularly well suited for applications with low latency requirements and intermittent connectivity, e.g., tools for real-time collaboration and mobile computing. They are no universal solution, however: the design of a CRDT is tightly coupled to the data model of the application. While simply data structures like increasing counters and grow-only sets are well understood and quite easy to implement, more advanced structures quickly get complex and tricky. Typical difficulties include that the overhead of additional metadata can grow very large and that effective garbage collection of old data states can be a hard problem to solve [LPS10].

2.3.3 Client-Centric Consistency Models

The following four consistency models, originally developed and described in 1994 by Terry et al. for the Bayou project, are basic expectations a client application commonly has for the behavior of a weakly consistent data store from its point of view.

These models are also known as *session guarantees*, wherein a session is an abstraction for a long-running sequence of read or write operations that a (mobile) client application may perform, while being only intermittently connected to the data store. In particular, applications can create multiple sessions to request different guarantees for different purposes from the same data store. They can be implemented in a practical and efficient way by using a version vector for each server. That version vector has to be increased for every write operation that is applied by the server [Ter+94; TS06].

Read Your Writes

Read Your Writes ensures that for a particular client and a particular data item, a read operation following an earlier write operation does never return a state before the write operation took effect. The client can always observe the effects of its *own* writes. The model does not constrain when the effects of a write by one client can be observed by any *other* clients.

A data store guaranteeing this consistency model cannot—in contrast to the three other client-centric models described below—provide total availability, it can only provide

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

sticky availability: As long as clients do not switch between different nodes, operations can continue to be processed on all non-faulty nodes. If a client switches nodes and a network partition prevents propagation of the write operation, it would not be able to observe the effects of its own writes on the previous node upon a read on the new node. Then, the only option to prevent that inconsistency is to not apply the write operation at all and stop being available in case of network partitions [Baib; Bai+13b].

Monotonic Reads

Monotonic Reads ensures that for a particular client and a particular data item, a read operation does never return an earlier state than the one a previous read has already seen—it may only go forward and return the same or a more recent state. The model does not constrain the order in which reads done by *different* clients observe a state.

A data store guaranteeing this consistency model can achieve total availability. It can do that by the read operations only returning the effects of writes which have propagated and are present on all nodes. Then, regardless of which node a client does connect to, it will never read an earlier state since the same writes have to be present [Baib; Bai+13b].

Writes Follow Reads

Writes Follow Reads ensures that if a client reads some data item and then writes to *another* data item, another client that reads the effect of the second write must not see an earlier state of the first data item than the one the first client has seen. This consistency model is also known as session causality: It is assumed that since the first client wrote data after having read data, a causal relationship between the two operations could exist. For example, the first client could have read some measurement value, decided that it is outside some acceptable range of values, and triggered an alarm condition as a result. In that case, it would be strange for another client to notice the alarm but still see the measurement value as perfectly fine within range.

The model is different from the two session guarantees described above in that it affects *other* clients outside the session. The session causality guarantee can be decomposed into two different constraints on write operations: An *ordering* constraint so that each write operation properly follows previous relevant write operations, i.e., is only applied when they have completed. And a *propagation* constraint so that all nodes and their attached clients can only see the effects of a write operation after they have seen all previous write operation effects on which it depends. For some applications, it can be useful to relax the Writes Follow Reads guarantee and only require one of these two constraints [Ter+94].

Monotonic Writes

Monotonic Writes ensures that for all clients and a particular data item, writes done by a particular client must follow previous writes of the *same* client. The model does not constrain the order in which writes done by *different* clients are applied in relation to each other.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Like with Writes Follows Reads above, it does affect *other* clients outside the session, and it can be decomposed into two separate *ordering* and *propagation* constraints [Ter+94].

2.3.4 FIFO/PRAM Consistency

First in, first out (FIFO) consistency, also known as *Pipelined RAM (PRAM)* consistency, ensures that for all clients (everywhere), the writes done by one *particular* client are observed exactly in the order they were invoked, i.e., the order of writes is preserved as if they went through a pipeline. It combines the constraints of Read Your Writes, Monotonic Reads and Monotonic Writes. The model does not constrain the order in which writes done by *different* clients are seen in relation to each other.

It was presented by Lipton and Sandberg in 1988 as one of the first described consistency models, in an effort to improve the performance of concurrent access on shared memory by different processes in a computer system. There, too, all processes have to see memory writes from one process in the order they were issued, but memory writes from different processes may be observed in any order. Like Read Your Writes, FIFO consistency is sticky available [LS88; Kina].

2.3.5 Causal Consistency

Causal consistency combines FIFO consistency with the session causality of Writes Follow Reads. It captures the notion that causally related operations should appear in the same order for all clients, regardless of which client invoked them. Operations which cannot be causally related, e.g., because they ran concurrently, may appear in different order to different clients. Like Read Your Writes and FIFO, causal consistency is sticky available [VV15; Kina].

2.3.6 Sequential Consistency

Sequential consistency is an even stronger consistency model. It implies that all operations appear to take place in the same consistent total order for all clients. However, in contrast to causal consistency, it cannot provide availability in case of network partitions: some or all nodes will be unable to continue running operations. This also implies higher latency under normal conditions [VV15; Kina].

2.3.7 Linearizability

Linearizability is one of the strongest single-object consistency models. In addition to the sequential consistency constraint of having all operations appear in the same consistent total order for all clients, it requires that order to correspond to the real-time order of the operations. Every operation appears to take place atomically and its effects are visible to all nodes *at the same time*. A good example where such a guarantee is useful is when some nodes interact with each other using a side channel outside the network of the distributed system [VV15; Kina].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Linearizability Checkers

To help test whether a concrete implementation claiming to provide linearizability actually works correctly, dedicated tooling exists. One example is the Porcupine checker [Ath17a], which implements an algorithm described by Horn and Kroening; Lowe [HK15; Low17]. Generally, an executable model of a system as well as a recorded history of an execution (all operation invocations and completions in their relative order) is required as input for a checker. Then, the tooling runs a decision procedure to determine if the given history is linearizable with respect to the model. That is, whether every operation appears to execute atomically and instantaneously at some point between the invocation and response. If any such point cannot be found without causing contradictions, the history is rejected as non-linearizable. Unless the executable model was defined incorrectly, the tested implementation the history is sourced from must be faulty. Of course, the opposite result of a linearizable history does not prove that the tested implementation is working correctly under all circumstances—however, with randomly injecting faults and running lots of randomized tests, some level of confidence can be attained. Even though linearizability checking is NP-complete, on small histories, it can work suitably well in practice [Ath17b].

2.3.8 Multi-Object Models

While the single-object models described above may need to consider multiple data items for their consistency guarantees (e.g., in *Writes Follow Reads*, see 2.3.3), each operation can only read or modify exactly one item. To modify multiple items, the client has to issue multiple distinct operations in sequence. Such a sequence is called a transaction, but if the data store does not provide transactional guarantees, undesirable results can happen which can be difficult for the application to deal with. These are similar to (but should not be confused with) the problems that can occur in relational databases. There, the acronym ACID (atomicity, consistency, isolation, durability) is often used to describe the desired characteristics the data store should provide.

Atomicity

Missing transactional atomicity can lead to seeing the results of only some, but not all intended operations of another clients' transaction. This state can be semantically invalid for the application and lead to severe problems due to violated invariants.

For example in Figure 2.9, a data store with two accessing clients is shown. Client 1 wants to swap two values stored under key a and b. To do that, it first gets the two values, and then stores them under their new keys. Then, client 2, with some bad "timing" with its query, gets the same value x for both keys a and b—the other value, y, temporarily seems invisible as if it were deleted, a *dirty read*. Were the two transactions running sequentially one after another, depending on the order, client 2 would either see $a = x$ and $b = y$, or $a = y$ and $b = x$, but never this intermediate state. Should a transaction

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

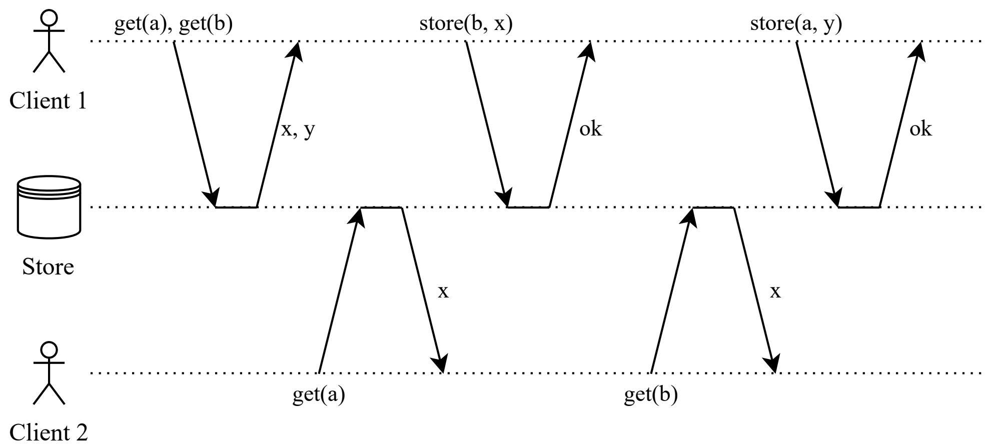


Figure 2.9: *Dirty read* timeline: C2 observes the effects of C1's half-finished transaction

fail to apply completely, the effects of all previous operations in the transaction have to be rolled back to satisfy the all-or-nothing semantics of atomicity.

Isolation

Missing transactional isolation from concurrent sessions can lead to different outcomes of the same operations, depending on whether the own transaction ran sequentially or concurrently with transactions of other clients.

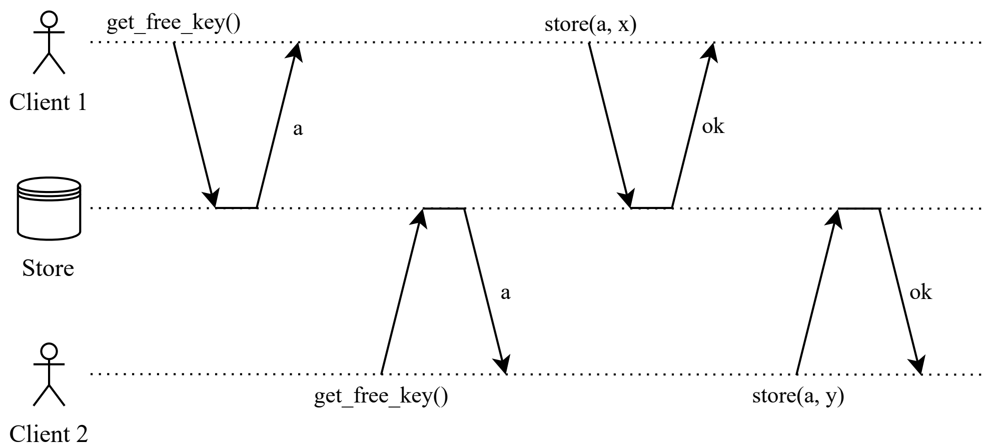


Figure 2.10: *Lost update* timeline: C2 inadvertently overwrites C1's value $a = x$

For example in Figure 2.10, a data store with two accessing clients is shown. Client 1 and client 2 both want to store some values x and y. To do that, they first query for a free key, then they store their value under the received key. Client 2 could end up inadvertently overwriting the value $a = x$ from client 1 with its own value $a = y$. This is due to both clients seeing the same key as free if both query operations run before

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

any of the store operations, causing a *lost update*. Were the two transactions running sequentially one after another, depending on the order, the result would be either $a = x$ and $b = y$, or $a = y$ and $b = x$, but there would be no loss of data.

In this simple contrived example, the store could anticipate that the clients want to store a value after fetching a free key and keep the returned keys reserved for some time to avoid collisions. In general, however, the store may not know what the clients are planning to do with the returned data and the clients may not know whether the returned data is still valid for use. Even if the data is immediately used for further operations, such as here, network delay and concurrency can lead to loss of data.

Highly Available Transactions

For the client to be safe from such concurrency artifacts, the distributed data store can implement consistency models that guarantee that such transactional or multi-object semantics are provided. Bailis et al. find that for less demanding transactional consistency models, it is even possible to provide high availability. They can continue processing operations without violating their consistency constraints during network partitions [Baia; Bai+13b].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 3

Related Work

This chapter discusses relevant research for our system design. We briefly consider consistency models, general principles for safe network communication and replication algorithms for synchronizing data in a distributed system.

3.1 Consistency Models

As introduced in Section 2.3, a consistency model is a contract some distributed data store can provide to its users. These models can be classified into two distinct groups: consistency models for non-transactional systems, where any operation can only affect a single object, and models for transactional systems, where an operation supports affecting multiple objects at once.

Viotti and Vukolic [VV15] created a comprehensive overview of consistency notions in *non-transactional* distributed storage systems. They formally defined 50 different consistency semantics from four decades of research. Additionally, Viotti and Vukolic established a partial order according to the “strength” of the guarantees of each model. See Section 2.3 for descriptions of the most significant consistency models and their relation to each other.

Bailis et al. [Bai+13b] considered the problem of providing Highly Available Transactions (HATs): *Transactional* guarantees that provide availability even during network partitions and that do not incur high network latency. Bailis et al. introduced a taxonomy and analyzed existing ACID isolation and consistency guarantees to identify which ones can and which ones cannot be achieved in HAT systems, and which semantic compromises may need to be made to do so.

3.1.1 PRAM/FIFO Consistency Verification

Wei et al. [Wei+16] focused on the verification of Pipelined RAM (PRAM) or FIFO consistency. They defined the Verifying Pipelined-RAM Consistency (VPC) decision problem and proved the algorithmic complexity of the general problem as NP-complete. Wei et al. also described an algorithm called Read-Centric for solving a variant in polynomial time.

PRAM Consistency

Wei et al. give the following definition of PRAM consistency: "A read/write trace satisfies Pipelined RAM consistency if and only if for each individual process, there exists a legal schedule of its visible operations, respecting both program order and write-to order." ([Wei+16])

In their formal model, an *operation* is a tuple containing the type of operation (read or write), the issuing process, the variable being involved and the value either being returned or being written to that variable. A *schedule* of a set of operations is a permutation of the operations with a defined precedence relation. There are two partial orders defined over the operations: The *program order* (PO) is the order in which operations are issued by each process. The *write-to order* (WR) associates each read with a unique preceding write (to the same variable) from which it reads the value.[1]

VPC Decision Problem

Using the definitions above, Wei et al. define VPC as follows:

INSTANCE A read/write trace T. Its size (denoted by n) is defined as the total number of operations in it.

QUESTION Does T satisfy Pipelined RAM consistency?

Wei et al. define four variants of the general VPC problem from two orthogonal dimensions, depending on whether there are multiple shared variables involved and whether the values written to those variables are unique (shown in Table 3.1). By reducing the strongly NP-complete 3-Partition problem to VPC, they prove that the variants that allow duplicate values are *NP-complete* (VPC-SD and VPC-MD). If only unique values are written to the variables, however, the problem is polynomially tractable (VPC-SU and VPC-MU). The reason is that with unique values, the corresponding (*dictating*) write for a read value can be identified deterministically.

[1] In other sections of this work, we generally refer to keys instead of variables, to data store clients instead of processes and to histories or observations instead of traces.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

	Single Variable	Multiple Variables
Duplicate Values	VPC-SD: *NP-complete*	VPC-MD: *NP-complete*
Unique Values	VPC-SU: *P*	**VPC-MU**: *P*

Table 3.1: Complexity classes for the VPC problem variants [Wei+16]

The Read-Centric Algorithm for VPC-MU

For the VCP-MU problem, Wei et al. provide a polynomial algorithm called Read-Centric. Since PRAM consistency is weak and does not require all processes to agree on a common view of order of the operations, each process can be checked separately. While write operations from all processes must be considered, read operations from other processes are invisible and can be ignored—only the *own* read operations are relevant.

The basic idea of the algorithm is to model the read/write trace of an observation as a directed graph. Operations are represented as nodes, while precedence relations are modeled as directed edges. While $o_1 \prec o_2$ denotes a *precedence relation* between the operations, $D(r)$ denotes the dictating write for the read operation r. To capture PRAM consistency, three rules must be applied:

Rule A (program order) For any pair of operations o_1 and o_2, if $o_1 \prec_{PO} o_2$, then add an edge from o_1 to o_2. This rule captures the program order for all pairs of operations issued by the same process.

Rule B (write-to order) For any pair of operations w and r, if $w \prec_{WR} r$, then add an edge from w to r. This rule captures the write-to order for all write/read pairs where the write operation is the dictating write for the read.

Rule C (w'wr order) For any triple of operations w, r and w' on the same variable, if $w = D(r) \wedge w' \prec r$, then add an edge from w' to w, leading to $w' \prec_{W'W} w \prec_{WR} r$. The rule captures the observation that in a legal PRAM schedule, there cannot be any other write w' between a read r and its dictating write operation $w = D(r)$.

After creating nodes for all operations, rules A and B are applied and a transitive closure of the graph is computed. Then, rule C is applied repeatedly as long as new edges are produced. Every time new edges are added, the transitive closure is also applied again. Once no edges can be added anymore, the algorithm is complete. Pipelined RAM consistency is satisfied if and only if the resulting graph is a directed, acyclic graph (DAG) [Wei+16].

These steps capture the RW-Closure algorithm with a worst-case time complexity of $O(n^5)$. The Read-Centric algorithm with $O(n^4)$ follows the same principle, but achieves better practical efficiency: Read-Centric processes read operations one by one and avoids redundant rule applications. Once the algorithm finds a certificate for inconsistency, it can terminate early, without having to collect all the operations.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

3.1.2 Bolt-On Causal Consistency

Bailis et al. [Bai+13a] described an algorithmic solution to upgrade an eventually consistent distributed data store to one that provides convergent causal consistency. That is accomplished by layering a "bolt-on" shim middleware on top, enabling reuse of existing and well-tested implementations and a clean separation of concerns. The bolt-on shim provides the consistency-related safety, while the underlying general-purpose data store can keep being responsible for liveness, replication, durability, and convergence.

Causality Tracking

As a prerequisite, the causal *happens-before* relationship between the write operations must be tracked (see also Section 2.3.5). There are two different ways to accomplish that: A data store can track *potential causality*, where for a write operation, all preceding write operations whose effects *may* have influenced that operation are taken as a causal dependency. If a pair of write operations has taken place concurrently, they cannot depend on one another. Future writes reflect any reads: After a process has seen the effect of a write operation, then all other processes must also be able to observe the effects of that operation before applying any future write operations of that process. The approach is conceptually simple and can be universally applied to all kinds of operations. However, the strong restrictions on the order of operations can lead to low throughput and high *visibility latency*. The visibility latency is the amount of time that the effects of a write operation must stay hidden due to the operation's dependencies being missing [Bai+12].

Bailis et al. choose to employ a different approach for their bolt-on consistency algorithm: That of *explicit causality* tracking, where the application must capture the causal relationships between operations on its own. Every write operation has to include a reference to all previous writes on which it causally depends upon. While explicit tracking places more of a burden on the application and there is a risk of missing causal relationships, the method can provide better scalability and throughput performance [Bai+13a].

Implementation

To provide causal consistency, Bailis et al. place a shim layer on top of each node of the underlying data store. All client operations are processed by the shim layer. The bolt-on shim tracks causality metadata and maintains its own local data store which is kept causally consistent at all times. Write operations are applied to the causal data store immediately. Then, they are forwarded to the underlying data store for distribution and durability. Processing of read operations, however, is more involved: While the underlying data store may permit a large variety of operation histories, the causality shim layer has to restrict the space of executions to those orders that do not violate the causal consistency model. There are two implementations available for read algorithms: an optimistic and a pessimistic one.

The *optimistic* read algorithm immediately returns the causally consistent value in the causal shim data store. While doing so, it queues the key that was read in a queue for

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

the background resolver. The resolver is an asynchronous background process responsible for fetching new updates, including their causal dependencies, from the underlying data store. As soon as all causal dependencies are satisfied, the updates are applied to the causal shim data store and become observable by clients. While this approach results in fast read operations that satisfy causal consistency, clients might experience poor visibility latency with unnecessarily stale values.

The *pessimistic* read algorithm, in contrast, does not return a value immediately. Instead, it checks the underlying data store for new updates to be applied. Since an update may causally depend on other updates, the algorithm has to check the data store recursively. It has to chase the entire dependency chain, resulting in slow read operations. For practicality, an implementation can mix both approaches. The read operations can trade off low latency with the staleness of values.

Causal Cuts

Bailis et al. employ *causal cuts* to determine which set of write operations is safe to apply. A causally consistent system maintains the invariant that each local store is a causal cut. Three rules guarantee that no causal dependencies are missing. A set of write operations is a causal cut if all of their dependencies are:

1. *Contained* in the causal cut themselves, or
2. *happens-before* to a same-key write operation already in the causal cut, or
3. *concurrent* with a same-key write operation already in the causal cut.

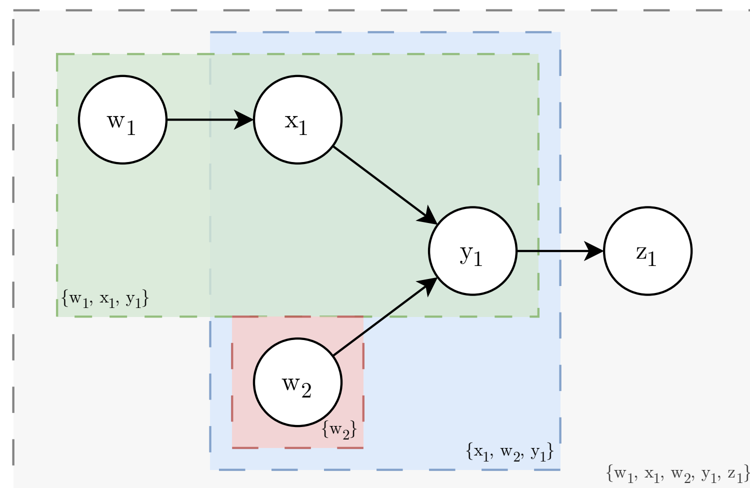


Figure 3.1: Examples of write sets that are causal cuts [Bai+13a]

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

As an example, consider the write history graph shown in Figure 3.1. Write operations to the keys $\{w, x, y, z\}$ are represented by nodes, with their causal *happens-before* dependencies represented as directed edges. The complete set of write operations is a causal cut, as are the subsets $\{w_1, x_1, y_1\}$, $\{x_1, w_2, y_1\}$, $\{x_1, w_2, y_1\}$ or just $\{w_2\}$. Sets with dependencies being missing are not causal cuts, e.g., $\{w_1, z_1\}$ or $\{y_1, z_1\}$.

3.2 Network and Safety Architecture

3.2.1 End-to-End Arguments in System Design

In their seminal paper, Saltzer, Reed, and Clark [SRC84] argued for a general end-to-end design principle for distributed computer systems. In a networked system using layered communication protocols, functions needed by the application have to be assigned to a suitable layer. However, it is not always clear which layer that is. Earlier systems were often designed to provide commonly required functionality—such as reliability and security—at lower networking layers as part of the communication system itself, only to discover that this was not sufficient.

Instead, the *end-to-end principle* arguments that functions which can only be correctly and completely implemented with the help of endpoints of the communication cannot be provided solely by the communication system itself. Doing so may be redundant or of little value, and should only be considered in exceptional cases. Saltzer, Reed, and Clark discuss a handful of typical examples:

- Ensuring data integrity on a file transmission
- Providing delivery acknowledgments to the sender of a message
- Encrypting the contents of a transmission
- Suppressing duplicate messages
- Guaranteeing first in, first out (FIFO) message delivery

Example: Data Integrity

Consider a file transmission from one host to another host. The underlying communications system may already ensure the integrity of packets between intermediary hops (gateways or routers), but that is not sufficient to ensure end-to-end integrity. Data can still be corrupted while being processed by an intermediate device or while being assembled by the receiving host. To detect such cases, an end-to-end data integrity mechanism has to be employed. For instance, a checksum of the contents can be calculated by the sending host and verified by the receiving host after assembling the data. In case the verification does not succeed, the transfer can be requested a second time.

Note that this functionality cannot be provided by the underlying communications system: application-level support is required. As soon as such a high-level mechanism is in place, hop-to-hop corruptions are detected, too—the low-level integrity mechanism is made redundant. The additional overhead may even prove detrimental to overall performance.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Sending everything again when only a few parts were damaged, however, is also not very efficient. Keeping the low-level mechanism in place may be useful for performance reasons. In such cases, carefully considered exemptions from the end-to-end principle may be justified. Overall, placing the functionality solely on application-level but designing it in a more sophisticated manner may still result in a more robust and scalable design.

Example: Delivery Acknowledgments

A similar argument can be made for a communications system that provides delivery acknowledgments for sent messages. Knowing only that a message was delivered to a target host is of limited value. What an application really wants to know is whether the message was successfully processed and some requested action was taken or not taken. For simple applications, the target processing can be made resilient enough that one can assume successful processing upon delivery. But the approach is not sufficient in case the processing depends on delivery to other hosts or can fail for other reasons, i.e., when a negative acknowledgment is a possible outcome. In that case, too, the real value lies in an application-level acknowledgment of successful or failed processing.

3.2.2 Congestion Avoidance and Control

Jacobson [Jac88] introduced many of the congestion avoidance algorithms used in modern TCP/IP implementations. At the time, the Internet suffered a series of *congestion collapses*. A congestion collapse is a dramatic drop of the data throughput between well-connected nodes due to network congestion. As cause for the problem, they identified the existing window-based transport protocol implementations behaving suboptimally and making the problem even worse with excessive retransmissions. As a remedy, Jacobson introduced seven new algorithms into the 4.3BSD (Berkeley UNIX) implementation of TCP and measured the improvements.

Conservation of Packets Principle

Jacobson's algorithms aim to adhere to the *conservation of packets* principle: A connection is "in equilibrium" when it runs stable with a full window of data in transit. Then, a new packet should not be put on the network until an old packet leaves, i.e., it is acknowledged by the receiver. Jacobson concludes that there are only three ways for packet conservation to fail, and treats each of these problems in turn:

1. To make sure a connection can even reach an equilibrium state, they use a slow-start algorithm that gradually increases the amount of data simultaneously allowed in-transit.
2. To make sure that a sender does not inject a new packet before an old packet has exited, they make sure that the sender's retransmit timer properly estimates round-trip time and implements an exponential back-off timer scheme for repeated retransmissions.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

3. In case an equilibrium cannot be reached due to resource limits along the path, they propose a *congestion avoidance* strategy consisting of two components: One component is that the endpoints must have a way to detect congestion and regulate utilization accordingly. Since packet loss due to damaged data is rare, they conclude that packet loss and timeouts are almost always due to network congestion and use that as a congestion signal. The other component is to ensure that the policies for window size decrease and increase are chosen carefully. In particular, the decrease must occur at least as fast as queues are growing (i.e., exponentially) and the increase is best performed slowly and linearly.

While the introduced algorithms solve the problem of congestion control for unicast connections, they are not suited for multicast transmissions. The reason is that they require the participation of the sender and the receiver in tandem.

3.2.3 Safety over EtherCAT (FSoE)

FailSafe over EtherCAT (FSoE) [Gro] is a communication protocol designed for implementing functional safety in industrial automation systems. The protocol is an open technology based on Ethernet for Control Automation Technology (EtherCAT) and part of the IEC 61784–3 international standard. FSoE can meet the requirements of Safety Integrity Level (SIL) 3.

Black Channel Approach

FSoE utilizes a *Black Channel* approach as main design principle: The underlying network connection does not perform any safety-related task and serves only as the transmission medium for safety data containers. The design principle has the advantage that the transport mechanism and the medium do not have to be included in the safety assessment. The safety functions are protected from compromise by a combination of measures within the FSoE safety protocol [Gro] (see also Table 3.2):

- Cyclic redundancy check (CRC) checksums guarantee that the contents of the received safety data containers are not corrupted.
- Watchdog timers guarantee that in case of any communication error condition, the devices reset to a safe state (fail-safe, typically upon exceeding 100 ms).
- Consecutive sequence numbers guarantee that any repetition, loss, our out-of-order delivery of safety data containers is detected and mitigated.

By employing the Black Channel approach, FSoE provides a level of flexibility and interoperability. Since the specification has no restrictions on the transport mechanism or medium, system design can be simplified and delivered with lower costs, while being

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Error / Measure	Sequence Number	Watchdog	Connection ID	CRC Calculation
Unintended repetition	✓			✓
Loss	✓	✓		✓
Insertion	✓			✓
Incorrect sequence	✓			✓
Corruption				✓
Unacceptable delay		✓		
Masquerade		✓		✓
Repeating memory errors in switches	✓			✓
Incorrect forwarding between segments			✓	

Table 3.2: Safety measures in Safety over EtherCAT [Gro]

robust, adaptable and maintaining the required functional safety standards. Safety-relevant data can share the infrastructure and travel on the same network as any other data.

3.2.4 Design of FoundationDB

Zhou et al. [Zho+21] described the basic design principles and architecture of FoundationDB, a distributed and transactional key-value store. These include transaction management, replication strategies as well as their approach for effective simulation testing. The software was open-sourced in 2018, is designed for horizontal scaling and offers strong guarantees for data consistency and durability. The transaction processing employs multiversion concurrency control (MVCC) for reads, optimistic concurrency control (OCC) for writes and provides the transactional consistency model of serializable isolation. While scope and goals of FoundationDB are very different from this work, we can learn from the design principles and the robust event-based testing.

Design Principles

The paper outlines four main design principles for FoundationDB: *Divide-and-Conquer* or *separation of concerns* decouples transaction management, the distributed storage and distributed configuration system from each other and allows them to scale independently. Within these systems, sub-systems are split up into separate processes that are assigned various roles, e.g., timestamp management, commit acceptance or conflict detection. Making *failure a common case* allows to reduce all failure handling to a single, well-tested recovery operation. The principle provides the assurance that even unforeseen or uncommon failure scenarios can be handled by the robust recovery operation. *Fail fast and recover fast* strives to minimize the Mean-Time-To-Recovery (MTTR), which

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

includes the time to detect a failure, shutdown the transaction management system and to recover. *Simulation testing* relies on a deterministic simulation framework to test the correctness of the implementation, both efficiently and with repeatability [Zho+21].

Deterministic Simulation Testing

As Zhou et al. put it, testing and debugging distributed systems is “at least as hard as building them.” Many sources of non-determinism can expose subtle bugs and break implicit design assumptions, leading to severe data corruption that may not be discovered for months. FoundationDB includes a deterministic simulation framework to end-to-end test the system and to reliably trigger and reproduce defects.

For the approach to work well, the data store system is fundamentally designed with simulation in mind: Instead of multithreaded concurrency, FoundationDB uses Flow, a C++ extension for asynchronous, actor-based concurrency. All sources of nondeterminism and communication are abstracted away and there are interfaces to inject unusual (but possible) behavior, e.g., to fail an operation that usually succeeds or to delay an operation that usually completes fast. The simulator runs composable, randomized workloads in a discrete-event simulation and injects various faults to test the system’s robustness. Among these faults, there are node crashes and reboots, as well as network faults such as network partitions and high communications latency. Conditional coverage macros are used to measure the effectiveness of the simulation and tune the scenario to increase the likelihood of achieving rare conditions [Zho+21].

Strengths and Limitations

The main advantage of discrete simulation testing is that defects can be found quickly. Bugs can be fixed early in testing instead of late in production, and can be traced back more easily to the related change that caused them. Discrete-event simulation can run arbitrarily faster than real time. By fast-forwarding the clock to the next event, the simulator can simulate long stretches of time. By running simulations in parallel, rare bugs can be triggered with a higher probability. However, the simulation is not suitable to detect performance issues. It is also unable to test code that does not support the simulator, such as third-party dependencies and parts of the code that are not implemented using the Flow extension. Bugs in critical dependent systems or misunderstandings of their contract can lead to issues not detectable by this kind of testing [Zho+21].

3.3 Consensus and Replication

3.3.1 Raft

Raft [OO14] is a consensus algorithm for managing a replicated log. The algorithm, presented by Ongaro and Ousterhout, produces a result equivalent to Lamport’s Paxos [Lam98], but aims to be easier to understand. That is accomplished by separating the key elements

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek Your knowledge hub WIEN

of consensus—leader election, log replication, and safety—and by enforcing a stronger degree of coherency to reduce the number of states that have to be considered.

Overview

Raft is a single-leader replication algorithm (see also Section 2.2.3): A designated leader node is responsible for managing the replicated log. The leader accepts append operations for new log entries, replicates them to other nodes and determines when it is safe for all nodes to apply them to their state machines. Raft divides time into *terms* that represent the reign of an elected leader node. Log entries always flow in one direction: from leaders to followers. Raft maintains the following five properties [OO14]:

- *Election Safety*: At most one leader can be elected in a term. Since a successful election requires a majority of the votes of all nodes, it can only result in one leader.
- *Leader Append-Only*: The log is immutable—a leader only appends new entries to the log, but never overwrites or deletes existing ones. Immutability simplifies the implementation and enables log matching.
- *Log Matching*: Logs that contain an entry with the same index and term are identical in all preceding entries. The property is maintained by two mechanisms: the leader never changing any entries in its log and the followers performing a consistency check. Followers only append a received log entry if the previous entry matches the expected index and term. Since the check is done for each individual entry, the entire resulting log must be identical.
- *Leader Completeness*: If a log entry is committed in a term, then that entry will be present in the logs of all leaders for all later (higher-numbered) terms.
- *State Machine Safety*: If a node has applied a log entry at a given index to its state machine, no other node will ever apply a different log entry for the same index. In conjunction with the previous property, all nodes will apply exactly the same set of log entries to their state machines, in the same order.

Leader Election

In Raft, each participating node is in one of three states: *leader*, *candidate*, or *follower*. To maintain its role, the leader must periodically send heartbeat messages to all followers. When these messages are not received anymore, a new leader has to be elected. Any follower node that detects this condition can become a candidate, vote for itself, and requests votes from the other nodes. If a majority of cluster nodes voted for the candidate, it becomes the leader for that term and starts sending heartbeat messages. If another node claims leadership or if no candidate achieves a majority, the term ends early, the candidate switches back into follower state and a new election starts. Randomized election timeouts help to prevent split votes and ensure quick resolution.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Log Replication

With a leader node established, normal log replication takes place: Clients can send their requests to the leader node, which appends entries to its log and sends them to followers. Once a majority of followers acknowledge, the entry is considered safely replicated and can be committed, i.e., applied to the leader's state machine. This ensures that even if some nodes are slow to acknowledge, fail to apply the log entry or do not receive it at all, execution is not delayed. If a follower's log becomes inconsistent due to crashes or failures, the leader forces the follower's log to match by overwriting mismatching entries with its own.

Safety

Raft guarantees that each node's state machine applies the same commands in the same order. This is achieved through log matching and the *Leader Completeness* property, which ensures that the leader contains all committed entries from previous terms. Raft uses the voting process to keep a candidate from winning if it has an outdated log. To eliminate the risk of a log entry being overwritten by a future leader, Raft never commits log entries from previous terms, even if they have been acknowledged by a majority of followers—once an entry from the current term is committed, previous term entries will be indirectly committed, too. Leaders retry indefinitely to send unacknowledged entries, which makes the protocol resilient to crashes and restarts of follower nodes.

Cluster Membership Changes

Raft handles membership changes through a two-phase transition. Directly switching from one cluster configuration to the next would be unsafe, since servers switch at different times—for a brief period, two leaders could be elected for the same term. First, the cluster enters a transitional configuration of *joint consensus*, where any node in either the old or new configuration can serve as leader, and log entries are replicated to both sets. A majority from both configurations is required to commit an entry. Once the transitional configuration is committed safely, the cluster fully transitions to the new configuration, which avoids the risk of split-brain scenarios.

Log Compaction

Raft manages the continuous growth of its log through periodic compaction. Ongaro and Ousterhout discuss two approaches: One method is *snapshotting*, where the entire state is written to stable storage, allowing nodes to discard any older log entries. As no cluster coordination is necessary, each node can take snapshots independently at any time. Followers that lag significantly behind may require log entries the leader has already discarded—in that case, they have to receive a complete snapshot from the leader to catch up. Alternatively, a continuously running *incremental* compaction spreads out the compaction process over time, reducing the impact on normal replication operation. The existing Raft protocol can already support an implementation using log-structured merge

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

trees (LSMs) [O'N+96], where the data can be compacted when merging it into the next level of the data structure.

Strengths and Limitations

Raft is a consistency algorithm with high understandability. With a single elected leader node, Raft can provide linearizable consistency. The safety has been verified using a TLA+ formal specification [OO14] and proven in practice in numerous implementations, e.g., in CockroachDB (a distributed SQL database), or etcd (a strongly consistent distributed key-value data store) [Etc].

However, to ensure the strict consistency model, read operations must go through the leader node, too. To avoid the risk of returning *stale data*, the leader may not simply return the data it already has. Instead, it has to communicate with its followers first to check which of its entries have really been committed and to make sure that it does not have to yield to a more recently elected leader. That results in a considerable latency for all operations and limits throughput. The algorithm is not tolerant of Byzantine faults: All nodes trust the elected leader to follow the protocol and to send data that is trustworthy and correct [OO14].

3.3.2 Merkle Search Trees

Auvolat and Taïani [AT19] presented an efficient way to implement state-based CRDTs (see also Section 2.3.2). At the core of their contribution lies a tree data structure, the Merkle Search Tree (MST), which is used to efficiently compare and reconcile the datasets stored on two different hosts.

Efficient Synchronization

An MST is a specialized Merkle Tree [Mer87] that combines three properties [AT19]:

- A given set of items has a unique and deterministic tree representation,
- the order of the inserted keys is preserved, and
- trees are always kept in balance.

Together with a collision-resistant hash function, these properties permit an efficient remote comparison of trees: By only comparing the root hashes, the equality of the whole dataset can be determined at once. If both instances contain the same items, the Merkle hashes of the roots will be the same. But if the hashes differ, the trees cannot contain the same items. Then, the comparison can continue to compare the hashes on the next lower level. Subtrees with the same hash can be skipped entirely. In this way, item ranges that contain differences can be narrowed down quickly. Only these ranges must be exchanged to synchronize the stored items. With a balanced tree, the number of required hash comparisons is logarithmic to the number of stored items.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

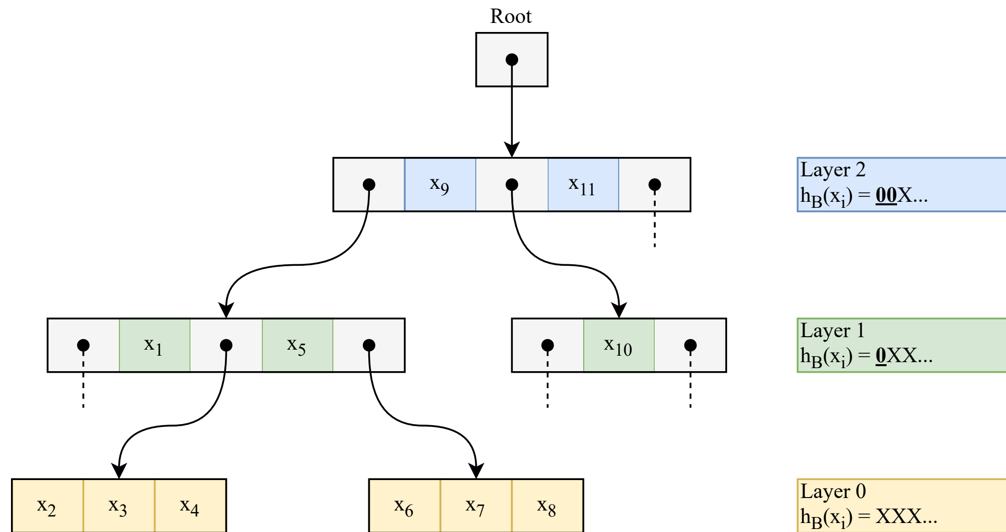


Figure 3.2: Structure of a Merkle Search Tree [AT19]

Deterministic Tree Construction

Auvolat and Taïani base the tree construction on a collision-resistant hash function. The hash function must have a negligible probability of finding two strings with the same hash, and it must project values uniformly. Modern hash functions such as SHA-512 satisfy these requirements. MSTs have a similar structure as B-trees, which are commonly used in relational database management systems (RDBMSs) for indexing. B-trees are a generalized, shallower form of binary trees, allowing for nodes to include pointers to more than two subtrees. This structure allows for faster traversal than binary trees. Typically, the node size is chosen to fit into a machine's memory page [AT19].

For MSTs, however, the shape of the tree is derived from the deterministic randomness of the used hash function (shown in Figure 3.2). The nodes are composed of a set of items that define how the space of keys is split, and of a set of pointers to the subtrees that contain the items for each interval in the split. To implement key-value maps, an item consists of a *key* that can be accompanied by a (CRDT) *value*. The subtree pointer consists of the computed hash value of the targeted node's contents (a *block*). The hash function has a secondary purpose: The tree layer an item will be stored in is determined by the zero-prefix-length of the computed hash value $h_B(x_i)$ (shown on the right hand side). If the output of the hash function is binary (Base $B = 2$), an upper layer contains about half the number of items of the next layer below. If the output is hexadecimal ($B = 16$), an upper layer contains only about one sixteenth of the items of the next layer below. The depth of the tree is about $\log_B n$ of the number of nodes. Item operations such as reads, puts and deletes can be implemented with a time complexity of $O(\log_B n)$. In combination with the order of the keys, the deterministic structure achieves *structural*

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

unicity: Regardless of the order that a set of items was inserted into the tree in, there is only one possible representation as an MST [AT19].

Strengths and Limitations

MSTs are particularly suited for replication in large, open networks where nodes may join and leave with a high frequency. Auvolat and Taïani use probabilistic gossiping for peers to synchronize their datasets and propagate changes. By using the same synchronization mechanism for replication, anti-entropy and recovery from faults, all three operations can share the same well-tested and robust code path. An additional benefit is that MSTs can provide causal consistency. This is due to a full state merge happening on every replication (gossip event) and the use of a separate buffer to hold partially received information until it can be merged in safely all at once.

The merge procedure optimizes for minimal bandwidth usage at the cost of higher worst-case replication latency: Several network roundtrips may be required to identify all differing blocks between two peer nodes. As a consequence of the probabilistic gossiping, there is no upper bound for the propagation delay until a change has been synchronized to all reachable nodes. There is also no built-in mechanism for delivery acknowledgment.

3.3.3 Bitmap Version Vectors

An alternative technique for replication and anti-entropy was introduced by Gonçalves et al. [Gon+15; Pet19]: Bitmap version vectors (BVVs).

Dotted Bitmaps

The central idea of the approach is that conventional version vectors can be enhanced with *dotted bitmaps*. These bitmaps serve as logical clocks, both for nodes and for the causality of individual items in the store. A *dot* represents a pair of a node identifier and version number, uniquely identifying every single write operation in the distributed data store together. The assigned version numbers are taken from a strictly monotonically increasing counter within the node. With these dots, every node keeps track of all observed write operations of every peer node. During a peer-wise anti-entropy synchronization, the nodes can efficiently compare their node logical clocks to retrieve all missing write operations.

The node logical clocks can be stored as BVVs, a compact representation as a vector of base/bitmap pairs, as shown in Figure 3.3. The *base* (green) represents all consecutive update versions up to the given version number, while the *bitmap* (orange) contains a single bit for every remaining distinct version number. The least significant bit is the first bit from the left. Each bit is a flag that stores whether the node has already received and processed the update operation with that version number issued by that peer node. Whenever a leading zero-bit (*gap*) of the bitmap is filled, the BVV can be compacted. The leading one-bits are then removed from the bitmap and added to the base counter. With that compact representation, the space the logical clock takes up depends on the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

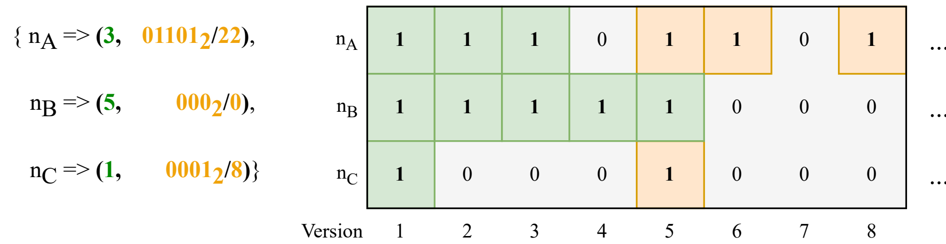


Figure 3.3: Node Logical Clock, implemented as Bitmap Version Vector [Gon+15]

number of peer nodes and the number of gaps, but is independent of the amount of past versions [Gon+15].

Causal Context

Gonçalves et al. also include a per-object version vector to track the *happens-before* relationship between update operations. One dotted causal container (DCC) is kept for every stored item. The DCC contains a map of dots to current values, as well as a version vector with past causal information, the causal context. Whenever a client issues a read request, it receives an opaque causal context together with the current value. By passing the context again on a subsequent write operation, the updated value is stored with a causal relationship to the read version. The data store uses the causal context to decide whether value updates supersede each other (one *happens-before* the other) or they took place concurrently with each other. Concurrently written values are kept until overwritten with a newer version. To querying clients, they are returned as a list—leaving conflict resolution to their discretion [Pet19, p. 249].

NDC Framework: DottedDB

Gonçalves et al. evolved the dotted bitmaps concept to the node-wide dot-based clocks (NDC) framework [Gon+17]. Here, BVVs are called dotted version vectors (DVVs) and DCCs are called NDC objects. The framework aims to solve scalability challenges associated with the high internal complexity of distributed key-value storage systems:

- *Conflict Detection*: efficient detection through the comparison of object logical clocks
- *Node Churn*: reduced metadata pollution due to leaving nodes by stripping their metadata from object logical clocks over time
- *Distributed Deletes*: efficient handling of deletes by employing the node logical clocks instead of depending on separate tombstone objects

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- *Anti-Entropy Repair*: efficient anti-entropy synchronization through peer-wise comparison of the node logical clocks and exchange of missing objects only

To evaluate the design, Gonçalves et al. built a prototype data store called *DottedDB* and tested it against *MerkleDB*, an otherwise identical data store that uses Merkle Trees instead. Both variants use Erlang and are based on the *Riak Core* framework for distributed systems [Gon16].

Strengths and Limitations

Compared to Merkle trees, DVVs require less computational effort, less bandwidth and only one network roundtrip to exchange metadata during an anti-entropy synchronization. Deleted keys and old value versions can be cleaned up fully as soon as the update has been replicated to all other replica nodes. Over time, the causality metadata stored in the *key logical clock* is slowly integrated into the advancing node clock. This has the advantage that for retired or unreachable nodes, the per-object metadata that has to be kept converges to zero [Gon+17].

The DVV concept is less well suited, however, for large, open networks, where there may be a high amount of node churn. The reason is that the metadata size of the *node logical clock* grows linearly with the number of nodes in the network—still reachable nodes as well as any nodes that participated in the past. The kept metadata for a past node cannot be cleaned (automatically). Should the node reappear, the metadata is required for efficient synchronization of the replicas [AT19].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

CHAPTER 4

Requirements

Towards answering Research Question RQ–1 for a suitable consistency model, we take a look at the system requirements for a Fire Detection and Fire Alarm System (FDAS) first.

Approach

Designing a modern FDAS system as a greenfield project clearly is too vast an undertaking for this work. The normative documents (e.g., EN 54 [EN 21]) seem to be written to match the properties of existing systems predating the standards instead of aiming for detailed and unambiguous specifications for new systems. Therefore, we follow a brownfield approach: We design a new data store component that can fit into the space of an already existing solution or be combined with other components.

First, we consider the decisions an existing solution took in the design space. Then, we determine functional and non-functional requirements for a new design, including a rationale for each item.

4.1 Existing Solution

This section considers an existing implementation of an FDAS DCU. The description is only a short overview, meant to convey additional background and to help inform the following phases of requirements gathering and system design.

4.1.1 Functional Overview

The existing FDAS solution is a distributed system supporting up to 16 member nodes. Each node consists of a control unit that functions autonomously, but shares event and

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

control data with the other control units. The system was designed with high reliability requirements and constrained hardware in mind. The design follows a highly integrated approach to optimize for low memory usage, storage capacity and link data rates. As a consequence, the design puts restrictions on modularity, interchangeability and future extensibility.

Distributed Event Data Processing

Whenever a notable event at a control unit changes the state of a logical entity, the node broadcasts a message to share the new state with all other control units. The receiving control units then apply the state to their local copy of the entity. This way, the data of the whole system is kept in perpetual synchronization.

In regular intervals, each unit runs a control loop that evaluates a set of pre-programmed rules. A stable snapshot of the data serves as input. A matching condition can trigger local events or set local state. The conditions take the form of boolean circuits, where logical operators such as *AND*, *OR*, *NOT*, etc. can be combined into bigger, reusable logic blocks. Most of these operators are purely functional and do not store any state (combinational logic [TS99, p. 655]). There are, however, some stateful operators available, e.g., event counters, edge detectors and flip-flops. Their output state does not only depend on the present input state, but on the sequence of past input states as well (sequential logic [TS99, p. 685]). The control rules are set on initial deployment of the system. Afterwards, they are usually modified only due to structural or regulatory changes.

Local Area Network Protocol

To transmit the messages, an application-specific LAN protocol is used for point-to-point connections between pairs of nodes. To achieve the required level of reliability, the protocol supports line, ring, and mesh network topologies. In addition, connections between nodes can be double-redundant, utilizing RS 485, optical fiber or TCP/IP overlay networking as the underlying link channel.

Flooding Routing Algorithm

To send messages throughout the local network, a flooding routing algorithm is employed. Each node acts as a router and forwards received messages through every outgoing link, except the one it arrived on [TW10, pp. 368–370]. Routing nodes keep track of forwarded messages to avoid relaying them a second time: Each message is tagged with a sequence number that is strictly monotonically increasing per originator node and channel. A routing node then stores the previously relayed sequence number per origin and only forwards messages with a higher sequence number. In case there is a gap between these numbers, the message is relayed but the counter is kept until the missing message is received. The algorithm ensures messages spread quickly and are delivered to all reachable parts of the network, but stop circulating eventually. Even unicast messages addressed to only one receiver have to be forwarded to all nodes to ensure they reach the recipient.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Link-Level Reliability

At the individual link level between nodes, care is taken to ensure reliable message broadcasting. For every message forwarded to a neighboring node, the forwarding node awaits an acknowledging response. There are three possible outcomes:

Acknowledgment The neighboring node has successfully received the message. As soon as all neighbors have acknowledged successful delivery, the forwarding node may remove the pending message from its buffer.

Busy The neighboring node cannot process the message at the moment. Typically, this happens because its message buffer is full. The forwarding node must keep the pending message in its buffer and retry delivery at a later time. This response is used to exert back-pressure, a mechanism we look at closer in the next Section 4.1.2.

No Response The neighboring node is not reachable. Either the network link or the node itself is faulty. The forwarding node must keep the pending message in its buffer and retry delivery for a while, but after a timeout it may give up and drop the message. In that case, the node broadcasts a fault message to inform the other nodes in the cluster of the faulty link or node.

FIFO Consistency

Every message is tagged with a unique tuple consisting of three elements: *(Sender identification, Channel number, Sequence number)*. The sender ID is the statically assigned node address within the system. The channel number specifies one of four available independent channels. For every channel within the sender, the sequence numbers for newly generated messages are assigned in strictly monotonically increasing order. With these sequence numbers, received messages are only made visible to the application once all the preceding messages for the same sender and channel have been received and processed.

This algorithm results in FIFO or PRAM consistency semantics [LS88] for each channel: Any pair of messages sent by a specific sender on the same channel are observed by every node in the order they were sent in. But messages from different senders or on different channels may be observed in arbitrary order (see also Section 2.3.4 on page 27).

4.1.2 Limitations for Scalability

The existing solution has proven its merits in the field and has been improved continuously. However, fundamental design decisions justified by the embedded systems available at the time may be limiting scalability on more capable modern hardware and covering additional use cases.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Flooding Routing Algorithm

Flooding is tremendously robust and simple to setup, but may be wasteful in case not all nodes need the information [TW10, pp. 368–370]. While it may work sufficiently well for line and ring topologies, the algorithm is inefficient for mesh topologies: A node can receive the same packet again and again from every one of its links. For the system as a whole, this means that a significant proportion of the network bandwidth and processing power is used for sending, receiving and then discarding packets with contents that are already known.

Hop-to-Hop Reliability

The existing solution ensures reliable transmission on every link between two nodes, but does not include end-to-end acknowledgment. Instead, it operates on the assumption that when every individual link is reliable (hop-to-hop), a message routed over multiple links will also be delivered reliably (end-to-end). That assumption, however, may not be true in every case. The end-to-end arguments made in Section 3.2.1 on page 36 support these observations:

- With no end-to-end confirmation of message delivery, the message sender has to trust that the network communication works well, with no low-impact recovery strategy being available if it does not. This approach demands a high degree of reliability for the message forwarding on the nodes themselves: If a node crashes in the critical time window between acknowledging a received message and waiting on the acknowledgment for the forwarded message, the message might get lost.
- Crashed or unreachable nodes in the cluster cannot be directly detected without a separate watchdog function. Their identity can only be inferred from the generated link fault messages, which also requires having static knowledge of the configured network topology available to draw the conclusion.
- All the synchronous acknowledgment response messages incur significant overhead for the system, in particular during normal operation without any message loss occurring. As the underlying hard-wired physical links typically have a very low loss rate, this is the common state. This is in stark contrast to shared medium technologies such as radio links, where hop-to-hop acknowledgment can be a useful addition to end-to-end delivery confirmation to improve overall efficiency. Additionally, keeping at most one message in flight while waiting for the acknowledgment severely limits the throughput of the links.

Back-Pressure Congestion Handling

The existing solution uses hop-by-hop back-pressure to handle congestion (see also [TW10, pp. 400–401]). When new messages arrive to the system at a faster rate than they can cross the network and depart the system, message buffers of the nodes are going to run

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

full eventually. With its buffers full, a node can only respond with *BUSY* to new message transmission attempts. This will cause the preceding node's message buffers to also run full, propagating the condition back towards the sender and exerting back pressure throughout the network. Since nodes are not allowed to drop any messages—unless a link is detected as completely faulty—the congestion eventually reaches and blocks all nodes from sending new messages. This may cause several unwanted effects in combination with other design decisions:

- By introducing even more messages into an already stressed system, retried transmissions and newly generated link fault messages can lead to a positive feedback loop. The feedback loop can further escalate the situation, with no mechanisms being available to detect and mitigate the congestion. Once the system has reached some tipping point, it can only stabilize again by nodes resetting themselves after some time without progress. The buffered messages may be lost in the process.
- Back-pressure congestion handling can impact messages indiscriminately: Messages from all senders or channels can be blocked simultaneously with no effective prioritization. During idle periods of low data rates, everything might be okay, but once the system is busy with high data rates, the behavior can result in problematic outcomes. A low-priority, but high-volume bulk transfer (e.g., a firmware update image) can easily delay or block messages of higher priority (e.g., a detector triggering a critical condition).
- In case the network topology contains loops, the congestion might lead to a deadlock when the four necessary *Coffman* conditions are met (*Mutual Exclusion*, *Hold and Wait*, *No Preemption*, and *Circular Wait*) [CES71]. All nodes might be blocked from forwarding their buffered messages indefinitely. Then, recovery is only possible by restarting nodes, interrupting links or purging pending message buffers.

By-Channel Consistency Semantics

With by-channel FIFO consistency semantics, causally related messages sent on different channels or by different senders can arrive in arbitrary order. Depending on the complexity of the programmed logical conditions, the effect can lead to confusing or incorrect results. For example, a node might detect a critical condition and trigger an alarm state. If both state changes are sent on different channels, a receiver might observe the alarm state itself before it can observe the condition that triggered it. When a condition only matches the alarm state, it should trigger reliably—however, if a condition expects the correct order of events, it might not reliably trigger.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

4.2 Requirements Analysis

Based on the regulatory framework and the scalability limitations of the previously described existing solution, we pursue a bottom-up approach to defining the requirements.

4.2.1 Overview

We want to gather the requirements for a reliable data store for usage in a Fire Detection and Fire Alarm System (FDAS). The system consists of a set of networked Fire Alarm Control Units (FACUs), which together form a dependable Distributed Control Unit (DCU). By writing to the shared data store, every node can share its current state and the state of attached detector and notification devices. By reading from the shared data store, every node can access the most recent states of the other nodes in the cluster and their attached devices. Using these operations, the control units can transparently utilize states of other nodes as source input for their programmed control loop.

Since an FDAS is a safety-relevant building system, the data store has to provide a high reliability. It must be highly available and fault-tolerant, i.e., be able to maintain proper operation in the event of failures or faults in one or more of its components. In particular, loss of network connectivity and crashes of individual nodes must be tolerated without reducing overall system availability and without losing any stored data.

4.2.2 Functional Requirements (FR)

Functional requirements specify what the system should do in terms of features and capabilities.

Data Storage and Querying

For the basic data storage and query capabilities, we can consult the compliance requirements listed in the EN 54–13 standard, which concerns itself with the assessment of system component compatibility and connectability [EN 19a]. In Section 4.3.5, which gives the requirements that apply when a network technology is used as transmission path between different system components, we find useful baseline provisions:

> "4.3.5 (d) if the configuration is designed to transmit a functional condition (such as fire alarm, fault warning, disablement. . .) from one CIE to any other CIE through the network, then the transmission time shall be determined by the applicable product standard. However, where this is not the case, transmission time shall be within 20 s;
>
> 4.3.5 (e) if the configuration is designed to transmit an activation message(s) from one CIE to any other CIE through the network, then the transmission time shall be within 20 s and the relevant output shall be activated at the other CIE as specified in EN54–2 or EN54–16;" ([EN 19a, p. 12])

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

For our purposes, the Control and Indicating Equipment (CIE) is equivalent to a control unit that contains a data store, and the given functional conditions and output activation messages are equivalent to the item values inside. Then, we can conclude that 20 seconds are a suitable upper boundary for the transmission time of value changes in the absence of faults.

FR–1: Limited Transmission Time

Changed data items must be readable on *all reachable nodes* within *20 seconds*.

Identifiable Source

Section 4.3.5 also has a provision about the traceability of significant functional conditions:

> "4.3.5 (f) if the configuration is designed to transmit a fire alarm, fault warning or disablement condition from one CIE to any other CIE through the network, it shall be possible to identify at least the CIE from which the information originated;" ([EN 19a, p. 12])

For our design, it is not clear whether this provision translates into a requirement for the data store or one for the application itself. While the data store likely needs to keep track of this information anyway, the application can easily solve this without involvement of the data store. It can include the node ID with the data item key or value, or establish a unique mapping by some other method. For instance, if each key is assigned an owner node and that is the only one allowed to write to it, the requirement is satisfied.

Failure Detection

In case of loss of communication to a network node, we can find the maximum allowed time interval until the fault must be reported:

> "4.3.5 (g) a loss of communication to a network node shall cause at least one CIE to enter the fault warning condition within 100 s. In the case of a hierarchical system, the main CIE shall enter the fault warning condition within 20 s of the original fault warning condition;" ([EN 19a, p. 13])

According to EN 54–01, a hierarchical system is "a system comprising more than one CIE in which a CIE is designated as main unit which is able to receive and transmit signals to any subsidiary CIE and to indicate the status of any subsidiary CIE" ([EN 21, p. 11]). For our design, we do not want to restrict the allowed configurations and use cases. However, since the two provisions overlap, we can satisfy both of them by mandating their union:

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

FR–2: Unreachable Node Detection

Unreachable nodes must be detected by *at least one* other node within *20 seconds*.

Fault Tolerance and Availability

Fault tolerance and availability are usually categorized towards the non-functional requirements. For a safety relevant system, however, these concerns are essential: We want the system to fulfill its function with high reliability. It has to tolerate link and node failures well, as described in Section 2.1. A distributed data store (DDS) can only provide availability when the data is replicated between nodes, as discussed in Section 2.2. Otherwise, one faulty link or non-responding node may render parts of the data completely unavailable until the issue is resolved.

FR–3: Continuous Data Replication

Data items must be continuously replicated across nodes.

In case of a component failure, the rest of the system should be able to continue to function. This requirement is at odds with strong consistency models (Sequential and Linearizable), as they cannot stay available in case of network partition—some or all nodes would have to pause operations to ensure consistency.

FR–4: High Availability

The remaining nodes of the distributed data store must remain available for operations in case of a component failure (network partition or node crash).

When reachable again, the replication process should continue to synchronize the contents of the data store. It must not lose any changes that were made in the meantime to the data items on either side of the partition, as this could lead to an incorrect operation of the control logic.

FR–5: Fault Recovery

When a node becomes reachable again after a component failure (network partition or node crash), changed data items must be synchronized without any loss of data.

Data Consistency

The rationale for requiring data consistency is simple in principle: when we apply the *same control logic* to the current state of the system (input state) on different nodes, we expect it to lead to the *same results* (output state). This means that all nodes have to have the same consistent view of the current system state (represented by the values within the data store). In other words, mere implementation details of the data

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

store, effects of the data replication and of the underlying unreliable network must not negatively impact the required functionality.

Under that condition, there is a relationship between the control logic and the consistency model: The more expressive the control logic is (allowed to be), the stricter the required consistency model must be to guarantee for the condition to hold. Or, inversely, the weaker the guaranteed consistency model is, the more limited the control logic has to be. A weaker consistency model, however, offers the opportunity of lower latency and better availability for the data store. This is the fundamental trade-off the PACELC theorem concerns itself with (see Section 2.2.5 on page 20). As these properties are among the goals for our design, we consider several consistency model candidates for their implications towards the control logic (see also Section 2.3 on page 22):

FIFO/Pipelined RAM Consistency Nodes are guaranteed to observe the writes of any other node in the order they were *issued in*. Control logic can depend on seeing one node's state changes in their correct order, which may be important to recognize signal edges or an alarm status that follows a critical condition. Since the correct order is a common expectation, the data store must guarantee the FIFO model at the least.

Causal Consistency In addition to FIFO, nodes are guaranteed to observe *causally related writes* from all nodes in the same order. Control logic can depend on seeing the correct order of causally related state changes, even if they happen on different nodes. The need for this model depends on two conditions: First, are there state changes performed on different nodes with a causal relationship to each other? For instance, a control rule that sets a local output state based on a foreign input state would meet the criterion. And second, does the control logic have to support arbitrary logical conditions that depend on the order of state changes from different nodes? The answer is not clear—one could argue that this kind of complexity should be avoided in any case. However, should the need arise, an existing data store can be retrofitted with a software layer ensuring causal consistency. Bailis et al. presented a suitable algorithm in "Bolt-on causal consistency" (see also Section 3.1.2 on page 34).

Sequential Consistency In addition to causal consistency, nodes are guaranteed to observe *all writes* from all nodes in the same total order. This model (as well as the even stronger linearizability) cannot be available in the event of a network partition: Some or all nodes can be unable to make progress with their read and write operations [VV15]. This drawback is undesirable for a safety relevant system with a requirement for high availability.

From weighing the different benefits and drawbacks of these models, we conclude that FIFO consistency is a suitable minimal requirement, with causal consistency being an optional upgrade path.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

FR–6: FIFO/PRAM Consistency

Data item changes issued by a specific node must be observed in the order they were issued in, by every node in the system.

Prioritization

Not all data items or state changes carry the same urgency. While functional conditions such as alarm, fault or disablement are critical to be transmitted as fast as possible, background and maintenance data such as sensor self-monitoring data, firmware or configuration updates are not. But with equal treatment, low-priority but high-volume messages can easily delay or drown-out high-priority messages. That scenario is known as *priority inversion*: The effectively achieved priority can end up being the opposite of the actually intended message priority.

Note that the requirement may be at odds with Requirement FR–6, the consistency model required above. While a priority mechanism aims to deliver a certain subset of messages earlier at the expense of others, a consistency mechanism's goal is to deliver the messages in a consistent and deterministic order. Unless these two order happen to match, the mechanisms may conflict with each other. For that reason, the consistency model has to be aware of priorities and tolerate missing messages adequately.

FR–7: Priority for Important Items

Changes to important data items should be replicated with priority. In case of network congestion, low priority items should be dropped first. With a priority-aware FIFO consistency model, missing updates of low-priority items should not delay delivery of high-priority data items (*priority inversion*).

4.2.3 Non-Functional Requirements (NFR)

Non-functional requirements specify criteria that can be used to judge the operation of a system, rather than specific behaviors.

Scalability

To be able to grow easily and be future-proof and evolvable, the system should have good scalability. For one, it should handle a growing network size well. We also want the ability to incorporate new functionality and to support a growing number of stored data items. Growths in these dimensions should not disrupt the expected functionality and should not negatively impact performance characteristics disproportionately.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

NFR–1: Functional Scalability

The data store should be able to handle a growing number of participating replica nodes and stored data items without disproportionately impairing the expected functionality.

We want to avoid designing a distributed system with a positive feedback loop in case of component faults. Error messages generated for delivery problems, for instance, should not lead to even more error messages generated when these, too, cannot be delivered. Otherwise, the system is at risk from making a dire situation even worse for itself, which can escalate quickly and surprisingly, disrupting even otherwise unaffected functionality. In the worst case, the system may get so unstable that only a hard reset can restore operation.

NFR–2: Fault Scalability

The data store should be able to tolerate a growing amount of component faults without disproportionately impairing the expected functionality.

Modularity

To keep maintainability and future adaptability high, we want to avoid creating a tightly coupled architecture. Instead, different concerns should be kept modular and separate where possible.

NFR–3: Modularity

Different concerns of the data store should be kept separate in the software architecture. Different components should be modular, encapsulate their internal details and have well-defined interfaces with minimal surface area.

Low Resource Usage

The individual nodes must run on resource constrained embedded devices. The resource requirements have to be appropriately low, including limited needs for processing power, memory usage and network bandwidth—see also Section 2.1.5 on page 12.

NFR–4: Low Resource Usage

A node has to run well on resource constrained embedded devices.

4.2.4 Non-Requirements (NR)

Non-requirements clarify non-goals for the design of the system. They are not exhaustive, but should help inform the design process and avoid unintended growth in scope.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Data Integrity and Authentication

Ensuring the integrity of received data is important for every system that communicates over a network, to protect it from accidental malfunctions and from deliberate tampering.

For one, a message's data content can be damaged or truncated in transit. To detect and discard malformed data, checksums such as Cyclic redundancy check (CRC) are typically calculated and verified by lower layers of the networking stack. The Transmission Control Protocol (TCP) and User Datagram Protocol (UDP) protocols, for instance, include a 16-bit checksum field in the header section. When communicating via such a protocol, there is no need for the distributed data store to include an additional checksum.

NR–1: No Integrity Check of Received Data

The data store can trust the received data and does not need to check its integrity.

For malicious manipulation of data a checksum is not sufficient. An adversary can calculate a new checksum that matches the modified payload. To protect from attacks, the sender would need to calculate and attach a cryptographic signature for the receiver to authenticate. Additionally, the required key material would have to be securely distributed to all legitimate nodes beforehand.

The EN 54 standards do not mandate authentication. They may trust that the physical security of the building is sufficiently high. If needed, authentication could be added as a separate lower networking layer. For instance, Transport Layer Security (TLS), Internet Protocol Security (IPsec), and other virtual private network (VPN) protocols provide mutual authentication and secure encrypted communication.

NR–2: No Authentication of Received Data

The data store does not need to securely authenticate the sender of received data.

A Byzantine fault condition, where a node does not follow the defined protocol, is hard to detect and mitigate. For instance, a node may "lie" and forward different values to different receivers. Or appear as functioning to one observer, but failed to another one. The remaining nodes would need to reach consensus on excluding the faulty node, while making sure that the nodes taking part in the decision are not faulty, too. Since the EN 54 standards do not mandate Byzantine fault tolerance, our design does not require a built-in strategy for resilience.

NR–3: No Byzantine Fault Tolerance

The data store does not need to be resilient to individual nodes not following the defined protocol.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Dynamic Group Membership

The nodes participating in the distributed data store are known upon deployment of the system. Configuring them statically avoids significant complexity in the design. Depending on the chosen consistency model, safely transitioning the membership at runtime without violating the consistency constraints even for brief moments is a separate challenge. The Raft protocol, for example, supports dynamic changes of the members of the group. To reconfigure its *cluster* at runtime, Raft follows a carefully designed two-phase transition process, where consensus on the membership itself is required at all times (see also Section 3.3.1 on page 40).

NR–4: No Support for Dynamic Group Membership

The data store does not need to support dynamic change of the set of nodes that make up the distributed data store. A statically configured, up-front set of participating nodes is sufficient.

Transaction Support

Transactions are operations which act atomically on multiple data items. They are applied either to all or to none of them, and observers should not see an intermediate state. Requiring support for transactions has severe implications on the design of the data store and its consistency model—see Section 2.3.8 on page 28 for some concerns that would need to be considered and addressed in that case. For the purposes of our FDAS application, however, we expect single-object operations to be sufficient. We only need to support read and write operations for single items.

NR–5: No Transaction Support

The data store does not need to support transactions, i.e., operations which act atomically on multiple data items at once. Single-item *read* and *write* operations are sufficient.

Multi-Writer Data Items

Unlike in other typical applications, we do not expect multiple nodes to share and write to the same data items. The items represent state bound to a single location: A value that was measured at a physical sensor, a condition that was triggered at a control unit. Even configuration or firmware updates originate from some central interface where a maintenance technician can connect their equipment.

Consequently, every data item can be assigned to a single owner node. Only that node may write new values, which are then propagated throughout the distributed system to other nodes' item replica. We expect the non-requirement to simplify the implementation of concerns such as conflict resolution, scalability, and data consistency. Should support

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

for multi-writer data items be required in the future, a mitigation strategy can be used: Every writer can write to a separate copy of the item (e.g., by prefixing the node ID). With that strategy, the application is responsible for keeping the values synchronized and resolving any conflicts.

NR–6: No Multi-Writer Data Items

The data store does not need to support multiple nodes writing to the same data items. It is sufficient that every item belongs to an owner node, which is the only one that may issue write operations.

4.2.5 Research Question RQ–1

What is the weakest consistency model that is suitable for the purpose of a Fire Detection and Fire Alarm System?

As discussed with Requirement FR–6, we determine FIFO or Pipelined RAM consistency to be sufficient to cover the basic use cases of an FDAS. Causal consistency, the next stronger model, could ease programming of the control logic for advanced conditions without weakening availability, but is by no means required.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 5

System Design

In Chapter 4, we defined a set of requirements for a distributed data store. In this chapter, we develop a corresponding system design in order to answer Research Question RQ–2.

Approach

First, we research existing approaches, algorithms, and technologies for similar problems (see Chapter 3). Next, we define the system model and the interface a client application can use to interact with the stored data. As described in Section 2.2, distributed systems involve technical trade-offs that have to be weighed carefully against each other. There are no optimal solutions that fit all applications equally well—instead, we consider the benefits and drawbacks to find a good match to the requirements. Finally, we develop a prototype implementation to evaluate the design.

5.1 System Model

The Control and Indicating Equipment (CIE) nodes that make up a networked FDAS need to share their state with each other. The state consists of the most recently observed input values of the attached sensors (detectors) as well as the most recently produced output values. An output value can represent the state of the attached actuators (alarm devices) or any internally calculated value. In the PLC world, this is also known as the "process image". The complete state image could grow quite large. Most values, however, are not expected to change all that often. To save network bandwidth, only changed or outdated values should be propagated.

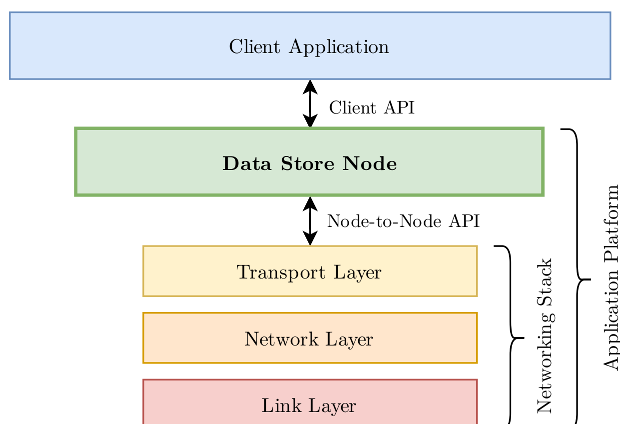


Figure 5.1: Layered Architecture: Platform for the Application

5.1.1 Key-Value Store

An effective way to divide the state image into smaller parts is to use a key-value paradigm. The individual input and output values can each be assigned to a unique and constant *key*. Choosing suitable keys is entirely up to the application. In our case, for example, it could be a single detector number, or a combined tuple of prefix, line number and individual element number.

The distributed data store consists of a set of nodes. Each node has its own storage, a unique ID and can communicate with other nodes by asynchronous message passing. Messages can arrive delayed, in different order or get lost entirely. Nodes can crash and restart, recovering their previous state from stable storage. Or they can fail indefinitely, and be replaced by a new node with a new ID and an empty state. Objects stored in the data store are replicated across all nodes.

5.1.2 Co-Located Clients

In a general distributed data store (DDS), a client can issue requests to any node of its choosing, and it may switch between different nodes. The latter, however, can have implications on the perceived data consistency.

For our design, instead, we expect the client application to always issue request to the same node (*sticky client*). Usually, the application will be co-located and run on the same physical device. In that case, the client-to-node latency is negligible and the availability for some consistency models can be better (*sticky availability*, see also Section 2.3). A few clients may not be co-located and query remote nodes instead, e.g., control panels or cloud API gateways. To maintain a consistent view, remote clients should refrain from switching between nodes, but the increased latency and reduced availability should be acceptable for these non-critical monitoring functions.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

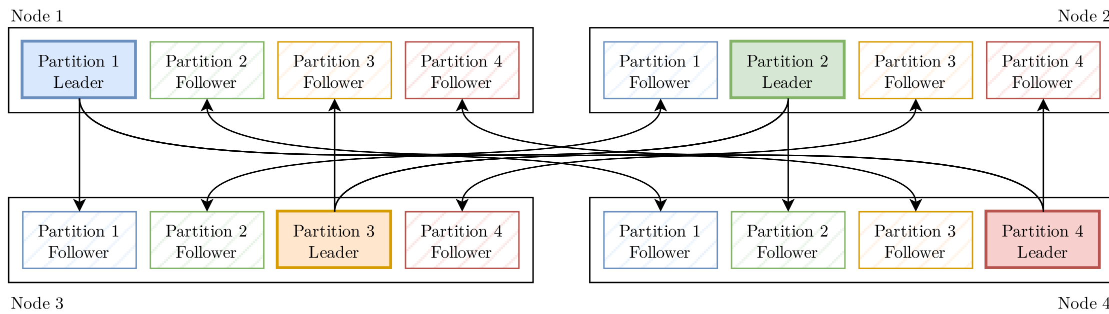


Figure 5.2: Replicated Data Partitions

5.1.3 Global Object Replication

In a general distributed data store using partitioning, objects are assigned to a partition that is replicated to a certain subset of nodes. If a node is queried for data it is not a replica node for, it must forward the request and wait for the results. This can result in considerable read latency for the client. In case of a network partition when none or not enough nodes are reachable, the request cannot be answered successfully.

For our design, instead, we mandate that the system always replicates all objects to all nodes. As a result, we can provide high availability for read operations: Regardless of network partitions or node crashes, as long as the client application can reach the local replica node, all objects are available in the most recently replicated version. Write operations also have good availability, as writing into the local store always succeeds. As soon as connectivity is restored, the update can either be sent out actively or synchronized by the anti-entropy process.

In general distributed systems terms, our design is a special case of replicated partitions, with the partitions being implicit and hidden to the client application (shown in Figure 5.2; see also Section 2.2.3). The data items are partitioned by the owning node. The owning node serves as leader node that accepts write operations. All other nodes serve as followers that maintain a read-only replica. In our case, the purpose of that setup is to maximize data availability while minimizing query latency.

5.1.4 Unreliable Networking Stack

From the underlying networking stack, we merely require best-effort delivery. In line with the end-to-end principle, we shift the responsibility for reliable operation upwards, from individual networking links and nodes to the distributed data store itself. We expect the approach to enable better scalability and adaptability. Additionally, we can benefit from the robustness of existing, well-tested protocols such as TCP or UDP.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.1.5 Client API

The database is a key-value store, where objects can be accessed by their key. To issue operations, a client first has to connect to the client API of its data store node and establish a session.

Session Parameters

The session does not start or manage any sort of transaction. Instead, it merely allows the client to configure the common parameters that affect all operations issued within the session:

- Consistency model: eventual, FIFO, or priority-aware FIFO
- Minimum priority filter: cut-off for priority-aware FIFO

Operations

A client can issue requests to a node using one of the following operations: It can read an objects' current value (*get*), write a new value (*put*), or query the set of existing keys within the store (*keys*).

- $get(K) \rightarrow V$
- $put(K, V)$
- $keys() \rightarrow \{K_1, K_2, \ldots\}$

Removal of items is not required—to free the space taken up by a large value, the application can simply overwrite the key with an empty value. While the expected size for keys and individual values is typically very small, the store has to support a significant number of keys without negatively impacting replication performance (NFR–1). Due to the sticky and usually co-located clients, the API can also more easily be extended later. Due to the global object replication, the set of existing keys can easily be queried locally and does not have to be collected from several nodes.

Extended Operations

It may be convenient for the application to be able to subscribe to data items of interest and be actively notified for any value changes (*subscribe*). The application developers may also desire a way to check successful delivery of a previously written value (*delivered*), which could also work subscription-based (observable) or require regular polling using a previously returned update identifier.

- $subscribe(K) \rightarrow Observable(V) \rightarrow \{V_1, V_2, \ldots\}$
- $delivered(U) \rightarrow Observable(N) \rightarrow \{N_1, N_2, \ldots\}$
  U . . . update ID returned by a previous $put(K, V) \rightarrow U$
  N . . . node ID

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.2 Dot-Based Clocks Framework

As starting point for our design, we select the node-wide dot-based clocks (NDC) framework by Gonçalves et al. [Gon+17]. It uses dot-based clocks, which are based on Bitmap version vectors (BVVs), to detect concurrent writes and to provide eventual consistency (see also Section 2.3.2 on eventual consistency and Section 3.3.3 on BVVs).

Merkle Trees could have been an interesting alternative mechanism for replication, anti-entropy, or both. They can efficiently determine differences between node replicas, as described by Auvolat and Taïani [AT19]. We decided not to pursue this approach: Merkle Trees use more computational effort to calculate the block hashes and can require several network roundtrips for synchronization. In addition, their strengths lie in managing a sizable network with high node churn. We do not expect that scenario for our application (see also Section 3.3.2).

5.2.1 Adjustable Parameters

In [Gon+17], Gonçalves et al. describe the replication and anti-entropy algorithms of the NDC framework. Several adjustable parameters and details are left for a concrete implementation to decide on, most notably:

- The assignment strategy of item keys to replica nodes,
- the lower-level details of client-to-node and node-to-node network communication,
- the replication and broadcast mode among the peer nodes, and
- peer selection and schedule interval for the anti-entropy background process.

The assignment of keys to replica nodes must be static and known to every node. For our prototype, we decide to distribute every item to all nodes. This strategy favors availability and latency: Client queries can simply be answered from the local store without requiring network communication. In the future, more sophisticated distribution strategies may be utilized. Replicating objects only to certain subsets of nodes could improve functional scalability and reduce resource usage on nodes, at the cost of an higher read latency and lower availability during network partitions. In case nodes regularly join or leave the system, a strategy to redistribute existing objects may be needed to ensure desired levels of availability.

5.2.2 Prototype Implementation

We started our prototype by implementing the NDC algorithms in C#/.NET 8. To debug the implementation and to verify that the functionality matches the expected behavior, we simulated all nodes in a single local process using async/await-based coroutines for all tasks. We ensured that the implementation behaves in the expected way by running some fundamental tests using the *NUnit* [Nun] unit test framework.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Networking Layer

To test the prototype using multiple connected nodes, we had to build a networking layer that supports the required remote procedure calls (RPCs). We chose well-established protocols that are suited for constrained nodes and are available as ready-to-use, open source components. The goal was to keep the required efforts for implementing and debugging the networking layer at a minimum. While these specific libraries are built for the .NET platform, they can easily be swapped out against their protocol-compatible native implementations which are well suited for embedded platforms.

- *protobuf-net* [Proa] for message serialization (*Protocol Buffers* compatible)
- *NetMQ* [Net] as lightweight messaging middleware (*ZeroMQ* compatible)

At first, we used the ZeroMQ *Request/Reply* socket pairs for the communication of clients to nodes and of nodes to each other. These are simple to start out with and to debug problems within the data store's communication logic, but their strict internal state machine is problematic for robustness: When request/reply messages are lost by the network or discarded due to a timeout, these socket types stay blocked in the waiting state and are unusable for any further communication. For RPCs where we need to wait for replies that contain the result of the call, we switched to *Request/Router* pairs, but configured the sending socket to use a relaxed version of the state machine and to correlate matching request/reply pairs using request identifiers. Otherwise, there is a risk of returning delayed reply messages that belong to earlier requests. For messages between nodes which we want to send-and-forget, we switched to the fully asynchronous *Dealer/Router* pairs.

With that, we are ready to extend the algorithms to meet our requirements' demands regarding replication of writes, the anti-entropy mechanism and the consistency models provided for the application clients.

5.2.3 Replication Mechanism

We require data items to be continuously replicated across nodes (Requirement FR–3) to ensure a high availability of the FDAS (Requirement FR–4) (see also Section 2.2.3). The NDC framework, as used in DottedDB, provides these for eventual consistency. It's protocol uses asynchronous message passing to send any changes to an item's values to all other nodes that replicate that key [Gon+17].

Asynchronous Replication Messages

The asynchronous, best-effort replication minimizes write latency at the cost of some risk of inconsistency, since there is no (immediate) acknowledgement mechanism. In terms of PACELC, with a low read quorum and/or high write replication count, the system can be classified as *PA/EL*: If a network partition occurs, it gives up consistency for availability. Under normal operation, it gives up consistency for lower latency (see also Section 2.2.5).

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

For our needs, asynchronous replication seems fine. We want low write latency and do not require immediate durability. For tracking of successful delivery, depending on the acceptable delay, we can either extend the replication with an acknowledgement mechanism or utilize the synchronization metadata which the anti-entropy process already keeps (see Section 5.4 below).

5.2.4 Anti-Entropy Mechanism

When a data store node becomes reachable again after a network fault or partition, it's replicated items must be synchronized to reflect their most recently written values (Requirement FR–5). To that purpose, the NDC framework includes an anti-entropy background process that runs periodically. It chooses a peer node at random and sends a synchronization request which includes the own node clock. That synchronization partner compares the received node clock with its own and sends back any missing objects for the node to apply (see also Section 2.3.2).

5.3 Consistency Models

Out of the box, the NDC framework only provides a weak consistency model: eventual consistency [Gon+17]. With eventual consistency, the store merges value updates on arrival, and they are immediately visible to client queries. This happens either by the normal replication process or by the anti-entropy background synchronization.

In addition, we want to provide the FIFO/PRAM Consistency (FR–6) with Priority for Important Items (FR–7). As all consistency models have benefits and drawbacks, the client application shall be able to select the appropriate model for every session. All operations issued within the session have to adhere to the selected model, shielding the application from arbitrary network behavior. The application *can* use several differently configured sessions concurrently, but its design should be careful about comparing or otherwise mixing the obtained data values.

5.3.1 Causal Context

The key to the framework's elegance lies in the *causal context* that is captured for every write operation (see also Section 3.3.3). To the client application, the causal context is an opaque object that simply is stored on reads and supplied again on subsequent writes. Internally, it consists of a version vector of past update IDs that shows for every pair of write operations whether they have a *happens-before* relationship to each other or if they were issued concurrently (see also Section 2.3.5).

If one write operation happened-before the other, its value is superseded and can simply be replaced. But if the write operations happened concurrently, all their values are kept and returned as a set on a *get* query. Conflict resolution between these values has to be done by the application. Upon the next write, when passing the previously returned causal context, these values are marked as happened-before the newly written single

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

value and replaced. Passing the causal context "confirms" that the concurrent values were seen and handled accordingly.

Single Writer Causal Context

With a single writer per item (Non-Requirement NR–6), we do not expect any concurrent writes or conflicts in regular operation. When supplying an empty causal context (or none at all) with a write, the NDC framework uses a minimal causal context that is always true: It simply marks all previous writes by the same node as happened before that write. Thus, unless we need to track causal relationships between writes from different nodes (to provide causal consistency), supplying the causal context is fully optional. It can be omitted from the client API and kept internal to the data store.

There is one exception: When an existing node is replaced or assigned a new node ID and writes new values to the same items, it has to supply the appropriate causal context to replace the previously written values. Otherwise, because of the two different node IDs, the new values are seen as concurrently written and the old values are kept (see also Section 5.6).

Merge Operations

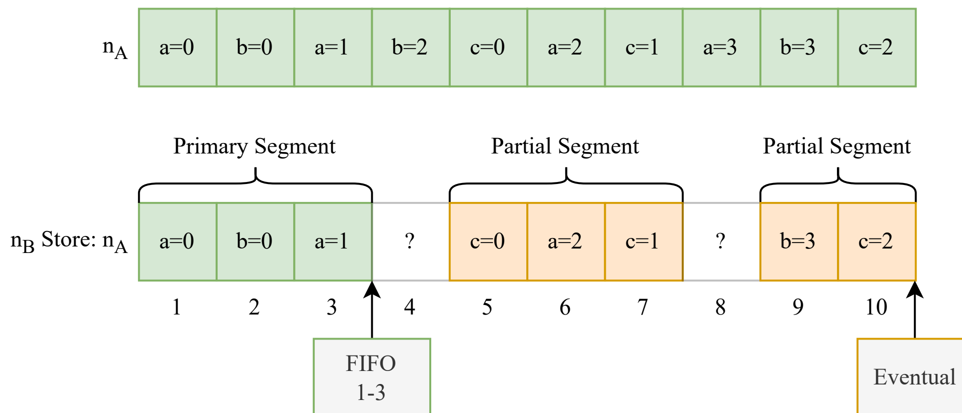
The causal context allows all NDC replicas to eventually converge to the same state, an essential property for eventual consistency (see Section 2.3.2). The merge operation is commutative and idempotent: The resulting value depends only on the set of write operations that were merged in, not on their order or whether they were applied multiple times. In this regard, the objects kept in the NDC Storage behave like state-based CRDTs (see also Section 2.3.2).

5.3.2 FIFO Consistency

Due to Requirement FR–6, we need the data store to provide the FIFO/PRAM consistency model. That is, the effects of writes issued by a specific client session (and thus, through a specific node) must be observed in the order they were issued in, by every client in the system. For writes to the same key, the merge operation already ensures that using the causal context—once replaced, superseded values can never reappear. Values written to different keys, however, can appear to a client in the arbitrary order the replication messages arrive in. For FIFO compliance, this visibility order needs to strictly correspond to the write order.

We can continue to use the NDC data structures and merge algorithms in principle, but we have to delay merging the received value updates until all previous updates issued by the same node are merged. Since the FIFO model has strictly more constraints than the eventual model, its visible value versions are always either equal or older than the value versions of the eventual model, effectively trading off the stronger consistency with higher latency.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



(a) Received updates from node A

Range	Role	Values
1–3	Primary	$a = 1, b = 0$
5–7	Partial	$a = 2, c = 1$
9–10	Partial	$b = 3, c = 2$

(b) Stored buffer segments for node A

Model	Effective Values*
FIFO	$a = 1, b = 0, c = \emptyset$
Eventual	$a = 2,\ b = 3,\ c = 2$

(c) Values by consistency model

Figure 5.3: FIFO replication data at node B

FIFO Replication

Since the NDC framework already keeps track of all seen writes (update IDs) within the Node Clock (NC), FIFO consistency can be implemented without changes to the replication protocol. We only have to handle missing updates differently. In case there is a gap between adjacent update IDs, we know that a write must have been issued by the respective peer node, but we know neither the written value nor the affected key.

To ensure consistent visibility order, value versions must only be made visible to a client application when the node has received and processed all previous writes issued by the same node, i.e., up to the first gap. The dotted version vector (DVV) entries kept in the NC are very efficient for that purpose: Value versions whose IDs are in the gap-less base section can be shown to the client, those in the bitmap section must stay hidden. This *FIFO condition* is easy to check and verify, but to be able to answer queries, additional data has to be kept in the FIFO specific storage. The reason is that the NDC framework merges in received updates immediately on arrival and keeps only the latest effective value versions in its eventual storage. When processing a query in FIFO mode, instead, we may need to return the values of earlier versions.

For example, refer to the situation shown in Figure 5.3a. Node B has only received eight

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

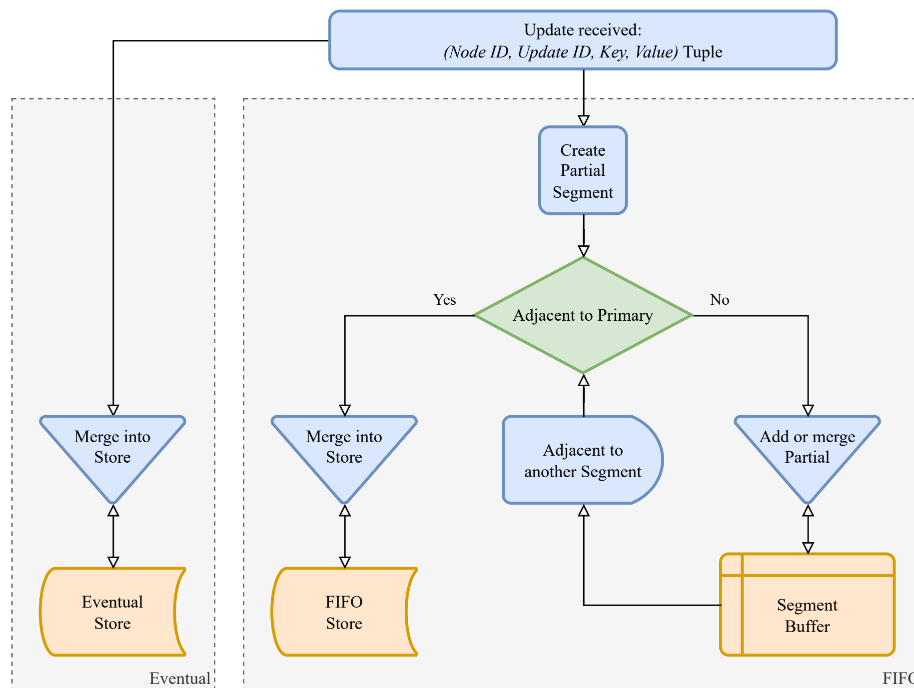


Figure 5.4: Flowchart showing FIFO update processing

of the 10 value updates issued by node A to the keys $\{a, b, c\}$. The eventual store has already merged in all received updates (Figure 5.3c). Simply returning its values could violate the FIFO condition: When a gap gets filled afterwards, we may show updates out-of-order. For gap 8, for example, $a = 3$ may appear *after* $c = 2$ is already seen. Notice that the shown effective values in Figure 5.3c are under the assumption that only node A has written to these keys—otherwise, the known primary segments of other nodes need to be merged in, too.

To solve this problem and keep the data available in the required granularity, we introduce a segment buffer per node within the store, as shown in Figure 5.3b. A segment is a contiguous range of covered update IDs, accompanied by their merged effective values. The primary segment has a special role: It covers the complete range until the first gap. As it is compliant at all times, its effective values can be—merged with the primary segments for other nodes—returned to clients querying in FIFO mode. The other partial segments hold buffered values that are not yet visible to clients and must be kept separate per node. Whenever an arrived update fills a gap, the adjacent segments can be merged together to form a larger segment. Their adjacent ranges can be combined, and their key values can be merged using the causal context. See Figure 5.4 for a high level overview of the data flow. For a more detailed description of our extensions to the NDC framework, see Appendix A on page 107.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

These segments have two significant benefits that keep FIFO order reassembly reasonably efficient: First, a node only has to keep as many segments in memory as there are gaps. As all gaps get filled eventually, only the primary segments remain, their combined value converging against the contents of the eventual store. Second, within every segment, only the most recent, effective value per key has to be kept. As segments can only be merged in full, other value versions are no longer required. In Figure 5.3a, for instance, the contents of updates 1 and 5 ($a = 0; c = 0$) can be dropped, as they are superseded by updates 3 and 7 in their respective segment ($a = 1; c = 1$).

For our proof-of-concept prototype, we store the segments in a sorted list and use a binary search algorithm to lookup segments which are adjacent to or contain the ID of a received update. To achieve better insertion and removal efficiency, a binary search tree could be utilized instead (amortized time complexity $O(\log n)$ instead of $O(n)$). The Linux kernel, for example, uses a red-black tree to store out-of-order packets for a TCP socket [Lin, `include/linux/tcp.h`, Line 245].

FIFO Anti-Entropy

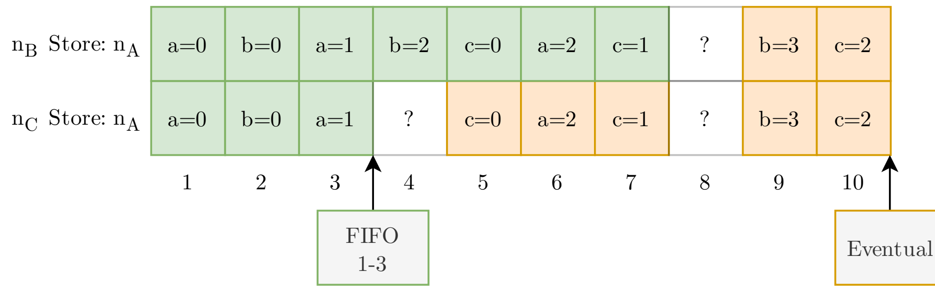
The FIFO replication can be implemented without any protocol changes. But the same is not true for the periodic anti-entropy synchronization between peer nodes. The reason is that the built-in NDC sync response only returns the resulting eventual value for each key where updates are missing, but not the individual updates that were merged in until that point. In case there were gaps in the node clock of the originating node, these values are not compliant to the FIFO condition and cannot be used. They cannot be split up again into the required segments.

For example, consider the situation shown in Figure 5.5a: The node under consideration, node C, has only received eight of the 10 updates issued by node A, updates 4 and 8 are missing and show up as gaps in the DVV. A client querying in FIFO mode can only see the effective values from the compliant segment, 1–3. A client querying in eventual mode, however, can see the merged set of all known value versions, as the visibility order and gaps do not matter.

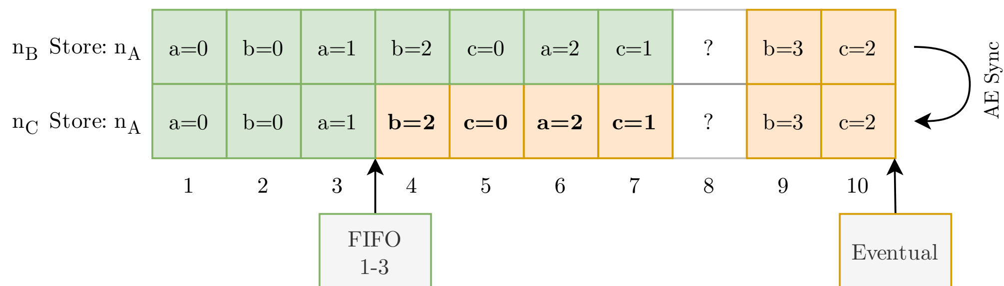
When node C sends an anti-entropy synchronization request to node B, node B compares the received node clock with its own—it has update 4, but update 8 is missing in its store, as well. Node B identifies update 4 as missing and sends a response with the resulting eventual value for the affected key, $b = 3$. Ideally, the new FIFO segment after the synchronization would comprise updates 1–7 instead of merely 1–3, but node C cannot safely apply updates 4–7 without also applying 9–10. In case the missing update 8 writes to some key other than b or c, the required visibility order would be violated (shown in Figure 5.5b).

While technically not violating FIFO consistency, this visibility delay is not desirable for a safety-relevant control system. A few lost update messages with unfortunate timing can lead to a large amount of value updates being hidden unnecessarily.

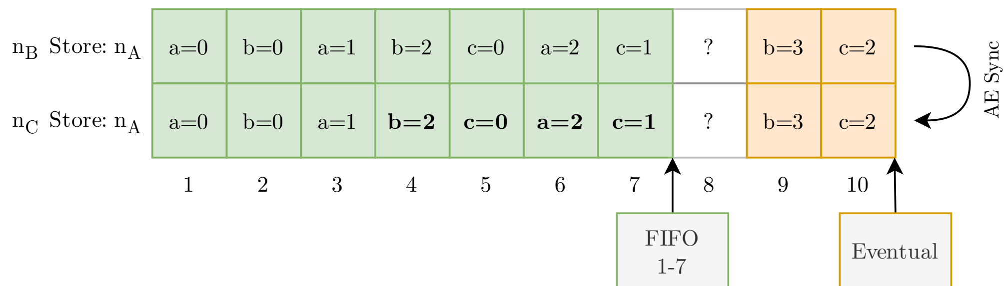
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub



(a) Initial state before anti-entropy: known updates issued by node A



(b) Anti-entropy without primary segment sync: node C still shows updates 1–3



(c) Anti-entropy with primary segment sync: node C shows updates 1–7

Model	Range	Effective Values
FIFO	1–3	$a = 1, b = 0, c = \emptyset$
FIFO	1–7	$\mathbf{a = 2, b = 2, c = 1}$
Eventual*	1–10	$a = 2,\ b = 3,\ c = 2$

(d) Effective values by consistency model and covered range

Figure 5.5: Lagging FIFO values without primary segment sync

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

To solve that problem and provide better Fault Scalability (Requirement NFR–2), we extend the synchronization protocol to include missing values from the FIFO specific storage, separately. The receiving node still only gets the effective value per key (the result from merging the primary segments per node). But since the source node is barred from merging in non-compliant segments, too, the node can trust the peer and merge it in (Figure 5.5c).

Notice that the effective values of the eventual model (shown in Figure 5.5d) are equal before and after the anti-entropy synchronization—the reason is that the missing update 4 ($b = 2$) is already superseded by update 9 ($b = 3$).

Synchronization of Buffered Partial Segments

With the extended protocol, synchronizing nodes only exchange missing values from the already FIFO compliant primary segments. Instead of a node being able to assemble partial segments by itself, it has to wait until one of the peer nodes was able to merge its partial segments. To avoid an unnecessary visibility delay, the buffered partial segments should also be included in the response to an anti-entropy synchronization request. Adjacent segments can then be assembled as soon as possible. The protocol optimization enables a faster and more robust recovery phase after any kind of network partition (Requirement NFR–2).

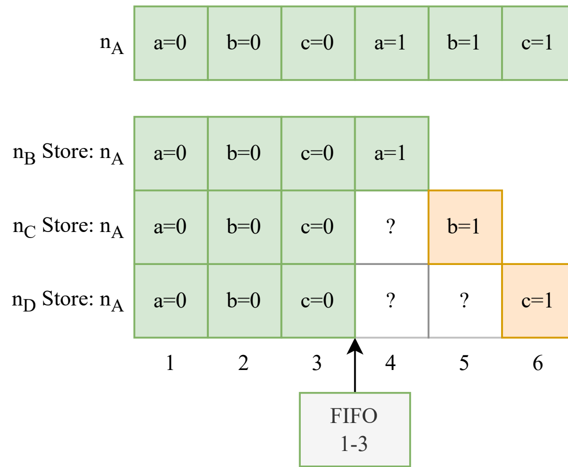
Suppose, for example, the situation shown in Figure 5.6a. Of the updates 1–3 from node A, nodes B, C and D have each only received one item via replication. For the first synchronization sync, node D selects node C as random partner. For the second sync partner, it selects node B. When synchronizing only the FIFO store, node D receives no updates in the first step, and receives only update 4 in the second. Due to the gap for update 5, the node can only show the range 1–4 to clients, 5 and 6 have to stay hidden (Figure 5.6b). To be able to show the whole range 1–6, the node would either have to select node A by chance, or in specific order node B and *then* node C.

With synchronization of partial segments, however, a partial segment for update 5 can be received in the first sync. After the second sync cycle, all segments for the range 1–6 can already be merged and shown to clients (Figure 5.6c). The random order of the anti-entropy synchronization has less chance to negatively impact the visibility delay.

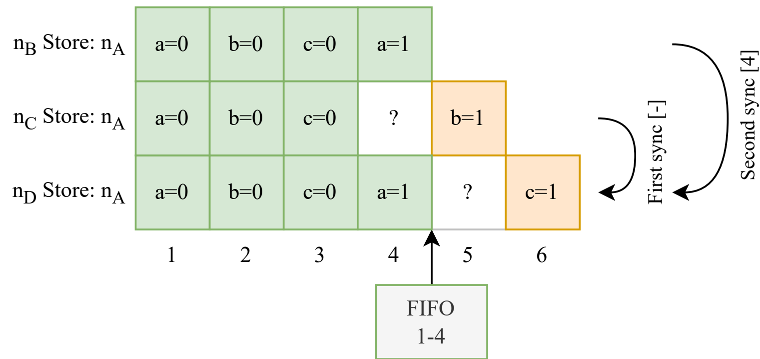
Invisible Intermediate Versions

Since version gaps can only be filled, never reappear, update IDs can only migrate from the bitmap section to the base section of a BVV. For the same node and consistency model, an older version of an item's value can never reappear after a newer version is visible. What can occur, however, is that when only one gap is filled by the anti-entropy process, several later versions can appear FIFO at once. Only the last effective version's value is returned. The querying client has no chance to observe the missed intermediate versions. The application design must take the possibility of such artifacts (time seemingly

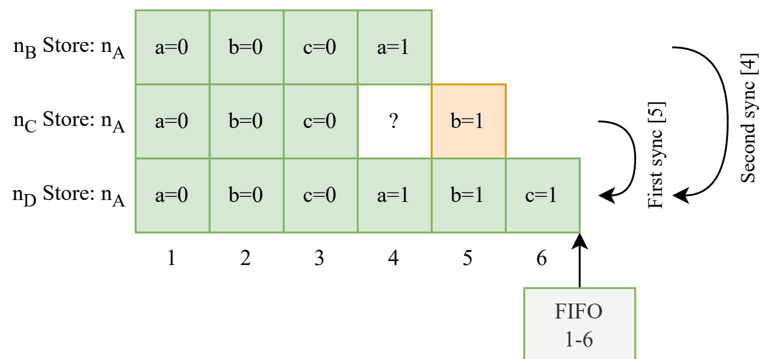
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



(a) Initial state before anti-entropy: known updates issued by node A



(b) AE without partial segment sync: node D only shows updates 1–4



(c) AE with partial segment sync: node D shows all updates 1–6

Figure 5.6: Lagging FIFO values without synchronization of partial segments

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

"skipping forward") into consideration and must not expect to see every single version without gaps.

To support querying missed values, the *get* operation would require some kind of cursor and return all versions since the last read. A subscription-based API like *subscribe* could indeed emit all missed versions' values in order. Whether this is worth the effort, however, is questionable. The semantic difference to the *get* operation and the possibility of receiving several values in very short order could increase the application's complexity and lead to interpretation errors that are difficult to debug. Such errors would only happen under specific circumstances after relatively rare network partitions. Therefore, the safest subscription design surely is to refrain from returning any missed values at all. Value changes should be emitted only after a replication or anti-entropy synchronization operation has run to completion.

5.3.3 Priority-Aware FIFO Consistency

Due to Requirement FR–7, the data store needs to be able to handle certain updates with priority in case of network congestion.

Priority Assignment

As a prerequisite, items within the data store and their replicated update messages must be assigned and tagged with the priority class they belong to. Within the data store, this can be as simple as recognizing well-known key prefixes, e.g., `P0-P3` for the four priority level classes $\{P_p \mid p \in (0, 3)\}$ in our prototype. Any other classification scheme can be used, as long as all nodes employ the same assignment and the classification of data items does not change over time.

Network Level Support

Many networking implementations take effort to avoid network congestion from happening in the first place (see Section 3.2.2 on page 37). However, if congestion does happen at any point on the transmission path, there are two different strategies: To not allow any messages to be discarded; or to build the system resilient enough that it can tolerate and recover from missing messages.

In the first case, once the forwarding buffer of a networking node is full, no further incoming messages can be accepted until the pending messages can be forwarded to the next node. This creates back-pressure that propagates backwards towards the sender. Unless the logical transmission paths for different priorities are entirely independent of each other, a congestion can block all messages equally. The strategy can be resource efficient, as the sender can immediately forget about a message once it is sent and acknowledged on link level—all responsibility for delivery is on the intermediate nodes within the network. But the approach mandates a high degree of control over the network and reliability from every single networking node. It can also severely reduce throughput

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

of the system, due to the overhead of all the link level acknowledgement messages. Some kind of recovery strategy has to be in place to maintain system functionality in any case (e.g., message loss due to a node failure).

An alternative that should scale better is to build a robust delivery confirmation and resending mechanism into the endpoints (following the end-to-end design principle, see Section 3.2.1 on page 36). With end-to-end acknowledgement, the system does not require link-level reliability and can tolerate networking failures better. The forwarding nodes are allowed to drop messages at any time. In case of congestion, the lowest-priority messages can be dropped first, and the freed capacity used to deliver more critical messages. To enable quality of service (QoS) traffic shaping with networking hardware—which cannot parse our custom payload format—the replication messages need to be tagged with their internal priority class. Tagging can be accomplished by assigning different VLAN priority tags (Ethernet, IEEE 802.1Q) or different destination ports per priority level (TCP/UDP). Details on these mechanisms are outside the scope of this work.

Data Store Support

As mentioned in Requirement FR–7, message prioritization is at odds with the goals of FIFO or stricter consistency models. Missing or dropped low-priority messages, e.g., from congestion control done by the networking stack, can affect delivery of high-priority messages. The mechanism enforcing a plain FIFO consistency may have to hide or hold back high-priority messages to avoid an inconsistently ordered view, working against the indented urgency.

We want to avoid this effect of *priority inversion*. The delivery of high-priority value updates should not be delayed only because some low-priority value updates are missing. We could enforce FIFO ordering only within each priority level and show updates that belong to different levels in arbitrary order. That model could, however, easily lead to obscure race conditions and application errors in case of any causal relationships between updates of data items of different priority. We want to shield the application from arbitrary behavior as much as possible.

To keep an intuitive expectation of FIFO ordering, instead, we introduce filtered FIFO views: When establishing a new session in FIFO mode, a client can choose the priority cut-off, the minimum priority level it is still interested in. If the lowest possible priority (*bulk*) is selected, the client will see all items and their value updates in strict FIFO order—exactly as before with plain FIFO. But when a higher priority level is selected, the client will only see items and value updates of that level or higher. With lower-priority items hidden, the data store can safely ignore their arrival order during enforcement of the consistency model and return the value updates of higher-priority items earlier.

These filtered views should provide a useful compromise between consistency and delivery latency. For the most critical items, however, eventual consistency is better suited. It can minimize latency by having updates visible immediately upon arrival.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

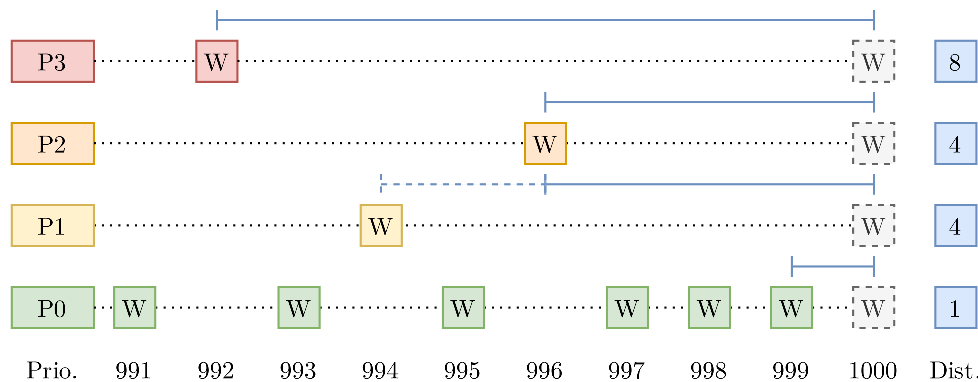


Figure 5.7: FIFO Priority/Predecessor Distances for Update 1000

Priority/Predecessor Vector

In contrast to plain FIFO replication, maintaining a FIFO consistency that is priority-aware requires extending the NDC framework's data types and replication protocol. That is because when there is a gap in the NC (an update issued by another node is missing), we have no way of knowing which data item that write operation has affected, and therefore, which priority level the missing update belongs to. If it is below our priority cut-off, we could safely ignore that gap and make the later updates visible. But if it is equal or above our priority cut-off, we must consider that gap and keep later updates hidden.

To solve that problem, we extend the *object* data structure that is kept in the NDC Storage (ST). For every key, it keeps a mapping of concurrent dot-values and the most recent causal context. For every dot/value pair, we add a priority/predecessor vector of previous update IDs per priority. That is, for each priority level, what is the immediately preceding update issued by the same node that has an equal or higher priority. In terms of distance, it indicates how many update IDs in between can safely be ignored—as they were assigned to lower-priority updates. All updates up until that predecessor ID must have been received without any gaps for the update to be applied and made FIFO visible. The higher the priority, the larger we expect the distance to be. Or, conversely, the smaller (the more far back) we expect the predecessor update ID to be. By storing the priority/predecessor vector together with the existing dot/value pair, their lifetimes are coupled, and the vector will be disposed by the existing NDC merge algorithm.

Implementation

With the priority/predecessor vector included with every single value update, extending the algorithm from Section 5.3.2 for prioritization support involves two steps. First, we maintain a separate, filtered FIFO store per priority level, which contains only values for items with equal-or-higher priority. By reading from the correct store, the client

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

Prio.	Predecessor
P3	Update 992
P2	Update 996
P1	Update 996
P0	Update 999

(a) Predecessor IDs

Prio.	Distance
P3	8
P2	4
P1	4
P0	1

(b) Relative Distances

Prio.	Distance
P3	8
P2	—
P1	4
P0	—

(c) Stripped Distances

Figure 5.8: FIFO Priority/Predecessor Vector Representations for Update 1000

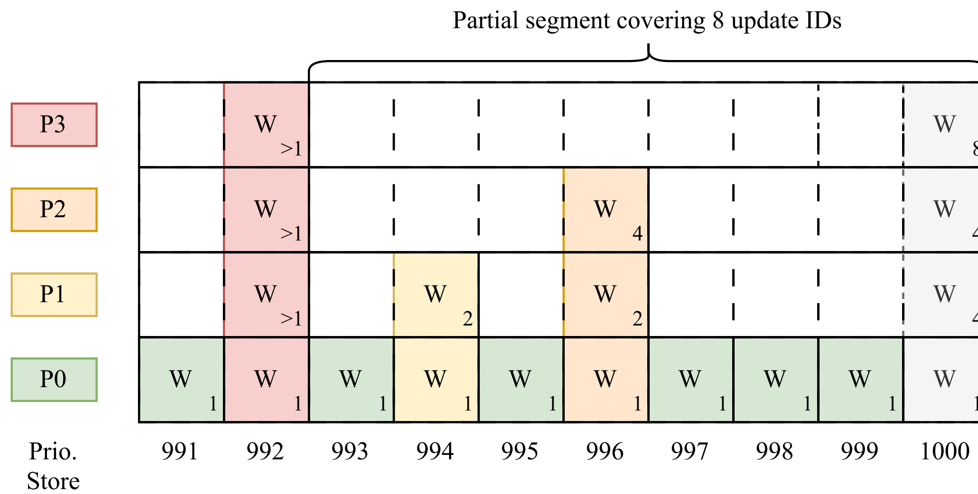


Figure 5.9: Partial FIFO segments per priority-specific store

session will receive results that match the minimum priority cut-off of its choice. When building and merging our global primary and partial segments, we ignore all updates of lower priority than the store threshold. For every inserted update, we look up the predecessor and add all update IDs in between to the partial segment's range, regardless of whether they were received or not. As a result, gaps caused by missing update IDs of lower priorities do not prevent merging of partial segments into the primary segment.

Figure 5.9 illustrates these partial segments with their minimal range determined by the priority/predecessor vector. Update 1000 arrives with the same vector that was shown in Figure 5.8. The vector values, indicating the distance to the priority-specific predecessor, determine the minimal length of the partial segment (shown as number in every writes' cell). It does not matter which key's value update 1000 actually writes to. Contrast this with Figure 5.3a on page 71 for general FIFO consistency, where partial segments could only cover an update range when every single update has been received. The lowest priority store *P0*, which contains the values of all keys, has exactly the same behavior, as

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

the distance to the predecessor is always 1.

Size Optimization

The priority/predecessor vector has to be included with every update operation and within the objects exchanged during the anti-entropy synchronization. To reduce the amount of data that has to be serialized and stored by every node, two optimizations are possible.

First, instead of including the complete predecessor ID, only the relative distance to the update ID can be given (Figure 5.8b). The distance numbers are smaller and can be efficiently encoded in ProtoBuf's Base 128 Varint encoding scheme [Prob]. Second, the distance to the lowest priority predecessor will always be 1—the write operation simply FIFO-depends on the immediately preceding write issued by the same node. Entries with this default distance can be omitted. Adjacent priority levels will often have the same distance. Since the distance of a higher level entry cannot be smaller than any lower level entry, for the same distance, only the lowest-level entry has to be kept—all higher entries can also be omitted (Figure 5.8c). The missing values can be back-filled on the receiver's side. This last optimization, however, is only beneficial with a substantial number of priority levels. In our example with four possible priorities, four values have to be sent in any case (distance array $[1, 4, 4, 8]$ vs. stripped distance map $\{P1 = 4, P3 = 8\}$).

5.3.4 Causal Consistency

Providing causal consistency is not one of the requirements we defined for the data store. If there are causal dependencies between write operations (value updates) processed by *different* nodes, the implemented application logic and the programmable control logic could benefit from them appearing in their correct causal order. With weaker models, the visibility order of causally related writes observed and issued by different nodes can be arbitrary. Note that due to its strictness, the latency between arrival and visibility of a value update can be even higher with this model than with FIFO consistency.

Causal Cuts Middleware

The NDC framework already uses a suitable causal context to distinguish between value updates with a *happens-before* causal relationship and those that happened *concurrently*. Thus, as a prerequisite, the application client needs to supply the proper causal context with every issued write operation, that is, the causal context that is returned with the previous, causally related get operation. With this partial ordering available, we could implement a visibility middleware that ensures operations are safe to apply using *Causal Cuts*, as described in "Bolt-on causal consistency" (see also Section 3.1.2) [Bai+13a].

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.4 Delivery Tracking

There are two ways to support the extended *delivered* operation and confirm successful delivery of updated values. First, we can use the node clock data that is already kept synchronized by the anti-entropy background process. Second, to provide faster confirmations, we could extend the replication process to actively acknowledge received updates.

5.4.1 Anti-Entropy Tracking

For the purpose of anti-entropy synchronization and efficient cleanup of fully replicated values, the NDC framework maintains a cache of other peer's states, as exchanged in the last round of pair-wise anti-entropy synchronization. We can compare the node/update identifier pair of interest with the kept data structures. When it's present within a peer's entry in the Node Clock (NC), we can confirm delivery to that peer. When it's present in the Watermark (WM), we can confirm delivery to all peers—the WM is used to determine when an update is present on all peers and can be safely removed from the Dot-Key Map (DKM).

One caveat of this convenient method is that the delay until acknowledgement depends upon the interval and peer selection of the anti-entropy process. To ensure a useful worst-case notification latency, these parameters need to be selected carefully.

5.4.2 Replication Tracking

To speed up the process and confirm successful delivery faster in the absence of network problems, the asynchronous replication RPC (*update*) could be extended to reply with an Acknowledgment (ACK) confirmation message. The reply can be processed asynchronously, without delaying the pending *put* operation the client is waiting on.

Since these messages are delivered on a best-effort base and can get lost or delayed at any time, this approach should only be used in addition to the anti-entropy tracking, not instead of it. Sending a separate ACK for every replicated value will also result in significantly increased network traffic and wasted bandwidth due to all the message overheads. Functional Scalability (Requirement NFR–1) should be kept in mind: To avoid self-limiting the update processing rate of the data store, ACK message could be broadcast in regular intervals and contain a BVV to batch confirm all updates received in between.

5.4.3 Application Level Tracking (End-To-End)

While these mechanisms can acknowledge delivery to a data store node, they can not confirm that an updated value was indeed seen and processed by the application. The changed value might not be queried by the client, or it might be made visible only with significant latency due to constraints imposed by the consistency model.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

For this reason, following the end-to-end design principle (Section 3.2.1) might be a good idea for delivery confirmation, too. Instead of relying on data store metadata, the application can simply maintain data items that signal observation of important changes. Doing so, however, can lead to updates written by different nodes causally depending on each other. Using a causal consistency model is recommended to avoid inconsistency bugs.

The usefulness of delivery tracking depends on its intended purpose within the application design. The application should not try to implement a retry mechanism—that is the responsibility of the data store. Instead of the pessimistic approach (tracking delivery of every single update) an optimistic approach (assume successful delivery, unless faults are detected) can provide better scalability for control and monitoring purposes.

5.5 Fault Detection

Fault conditions such as faulty links or unresponsive nodes must be detected (Requirement FR–2). Such reports are used to alert the staff on site and logged for upcoming maintenance work. They might also serve as input for the programmable FDAS control logic, e.g., to close fire doors by releasing their magnetic door holders as a precaution. There are multiple different ways to detect faults, which can also be combined.

5.5.1 Link Level Detection

Network links can detect that their direct partner is not available or has not sent anything in a long time and generate link-fault messages. Static knowledge of the configured network topology is required to infer the faulty or unreachable nodes. The approach (alone) is inflexible and cannot provide end-to-end monitoring.

5.5.2 Data Store Metadata

As the data store already needs to keep track of the state of other peer nodes, similar to Section 5.4 above, existing anti-entropy metadata can be used. While the replication mechanism is abstractly defined as using asynchronous message passing, it can be implemented so that connectivity errors, e.g., from lower layer protocols such as TCP, are reported.

In contrast to link level detection, this method at least provides node-to-node fault monitoring, and there is no static knowledge of the network topology required. The approach can be efficient, as no additional network messages are required. The same caveats apply, however: The timeout after the last successful connection must be carefully considered. For the pairwise anti-entropy synchronization, for example, there is a trade-off: To save network bandwidth, the interval should not be too short. Otherwise, a large amount of traffic are useless messages which do not yield any missing items to synchronize. To detect faults within a short time, however, the interval has to be kept

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

short. The approach mixes different concerns and introduces some dependencies between architecture layers.

5.5.3 Application Level Fault Detection (End-To-End)

Similarly to delivery tracking, relying on data store metadata alone cannot detect faults, hangs or crashes of the application. Instead, the application can implement its own mechanism to detect faulty or unreachable nodes. For instance, nodes can regularly broadcast a heartbeat signal to prove their healthiness using the distributed data store. They can write their uptime, wall-clock time or any other regularly changing value to a well-known key. Other nodes can periodically read the replicated value from their own data store and compare the value with a previous version. The result shows that the node itself is healthy and that the replication messages have successfully propagated.

Comparing heartbeat signals obtained from sessions with different consistency models can be useful: While a positive eventual signal shows node and connection healthiness at the moment, a positive FIFO signal confirms that the replicated updates (for that specific priority level) are up-to-date, too. A lagging FIFO heartbeat signal hints that other updates from the node may be lagging, too.

Note that this mechanism is unidirectional: A node can determine the liveness of another node, but it does not know whether its own heartbeat signals (and thus, replication messages) are received by others. A bidirectional, roundtrip confirmation may be beneficial for a safety-relevant FDAS application: Upon detecting isolation from the rest of the DCU, the node could switch its control logic to a better suited mode for autonomous operation. In principle, the causal context accompanying the heartbeat signal could be used to include the node clock of the sending node. The heartbeat signal would not only confirm liveness, but also show a baseline of the updates successfully received up to that point.

5.6 Storage Durability and Crash Recovery

For our NDC-derived store, each node continuously maintains five data structures. These are used to answer client queries, apply received replication updates and perform anti-entropy synchronizations [Gon+17]:

- Node Clock (NC): a set of all dots from current and past versions seen by this node (a DVV per node);
- Dot-Key Map (DKM): a mapping of locally stored data item version dots to the keys they belong to (required by the anti-entropy protocol);
- Watermark (WM): a cache of node clocks from every peer (required to know when a dot is present in all peers and can be removed from the DKM);

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

- Non-Stripped Keys (NSK): a set of local object keys whose causal context is not yet empty (required for the regular causality stripping background task);
- Storage (ST): a mapping of keys to objects, which contain the item value versions that are not yet obsoleted.

With these five data structures, the NDC framework implements eventual consistency very efficiently: Old value versions and object metadata that are no longer required are removed as early as possible. While NC and ST keep essential replication and value data, DKM, WM and NSK keep additional metadata to facilitate efficient anti-entropy synchronization and cleanup.

To answer queries with priority-aware FIFO consistency, additional sets of these data structures are needed. Instead of just keeping the most recently known value version, as sufficient for eventual consistency, they must preserve the value versions that are guaranteed to be made visible in FIFO order. If there are no replication updates missing and thus no gaps, these value versions are equal—but if there are gaps, the FIFO version can be older than the eventual version (see Section 5.3). Per supported priority level, we need to keep an additional separate instance of the store, consisting of a tuple of NC, ST, DKM, WM and NSK.

5.6.1 Durability

The node's own clock entry within the NC should be stored using durable, non-volatile memory, where it can be recovered from in case of a node crash. The other data structures can be kept in ephemeral memory as their most recent contents can be learned back from other nodes.

A hybrid approach is also feasible: Keep the most recent state in memory and, using an additional background task, periodically flush a consistent snapshot to durable storage. With this trade-off, regular operations can be processed quickly in-memory and wear on the storage media is reduced. In case of a node restart or crash, the majority of the data can be reloaded from storage instead of relying on the slow synchronization over the network.

5.6.2 Unknown Node Clock: Write Impediment

With an unknown value of the own node clock (the last assigned update ID), the successor update ID and the causal context for the next write cannot be reliably generated.

Simply using an approximate clock is problematic: If the used clock is too advanced, some update IDs between the real and the used clock are never assigned. These IDs represent gaps that can never be filled, as no node is able to provide their values during anti-entropy synchronization. For the eventual storage, these gaps do not impact query functionality. For FIFO specific storages, these gaps prevent each node from making progress and result in stale values being returned for queries. But if the used clock is

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

too far behind, some update IDs between the used clock and the real clock are used twice, with different items or values. These duplicate update IDs can lead to lost or inconsistent values between different nodes, at least until the node clock has caught up and the update IDs are unique again.

While we should take effort to prevent it in the first place, there are two strategies to recover from the loss of the node clock: We can try to learn back our own previous clock from other nodes, or assign a new node ID so that the node clock can simply start from zero again.

Best-Effort Clock Recovery Strategy

We could try to learn back our own lost clock entry by doing a few anti-entropy synchronization process rounds with other peers. Due to the eventual nature of the data store, however, we cannot reliably determine when we have *fully* caught up, and when it is *safe* to start processing client's writes. We can only learn a minimum value, which might be too low and run the risk of creating duplicate update IDs.

To mitigate that risk, we can add some safety offset to skip ahead—missing update IDs are safer than duplicate ones. To avoid producing stuck FIFO stores due to globally missing update IDs, we should issue the first write with a generous priority/predecessor vector that allows the priority stores to skip any gaps (risking some FIFO consistency violation for a short amount of time). With careful selection of the parameters of synchronization quorum and safety offset, this recovery strategy could work sufficiently well.

New Node ID Strategy

As an alternative, we could "forget" our old node ID and assign a new one. With a new, initial node clock, we can immediately start processing writes. To write to the same data items, we just have to be careful to include the old node ID with the highest possible update ID within the causal context. Otherwise, the new items will be detected as concurrent: The new versions will be merged with the old ones instead of replacing them.

There are three preconditions, however: The node needs to be able to allocate a unique and unused new node ID, without the risk of collisions with other nodes. Peer nodes must be able to learn of the new node's existence and be able to reach it afterwards for replication and anti-entropy rounds. Additionally, other nodes must be able to learn that the old node is gone and cannot be reached anymore, perhaps by marking it as replaced or by giving up connection attempts after some amount of exponential back-off cycles.

As we did not plan for dynamic cluster membership (Non-Requirement NR–4), all three mechanisms would need to be implemented first. The last one could take effort, as an unreachable node prevents the NDC framework from removing some associated metadata. A node that never returns can cause a memory leak.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

5.6.3 Missing Data: Read Impediment

Like for the unknown node clock above, for lost values of data store items, we can learn them back over time from peer nodes. But we cannot reliably determine when we have *fully* caught up. To avoid returning value versions that are too stale, it could be a good idea to refrain from answering client queries until successful synchronization with some quorum of peers was successfully performed.

5.7 Large Value Objects

An exceptional case are large objects shared using the data store, e.g., config packages to deploy or firmware update blobs to install. There are two different key mapping strategies.

5.7.1 Key Mapping Strategies

The whole object can be put into a *single item*. While simple for the application, the data store has to be able to support replicating large values. Transmitting the whole value at once may run the risk of blocking other items of higher priority. The value would need to be split into small chunks and transmitted in well dosed batches, an assembled to a whole object again once all chunks are received.

Alternatively, the object can be mapped to a *range of keys*, e.g., the target addresses where firmware blocks should be written to. In that case, single-item *put* and *get* operations may turn out to be an inefficient bottleneck. Operations which can efficiently read or write a whole key range may be required. Also, without support for transactions that group the writes together, keys with their data segments may appear to a receiving client in any arbitrary order. Transmitting a high number of changed items at once can also still easily overwhelm link capacity or replication mechanism and block higher priority items.

In addition to the mapping strategy, there is the general unicast-over-multicast and storage problem: When putting their value into data items, blobs whose contents may only be required by one or a few nodes need to be replicated to all replica nodes. Keeping the network load low requires an efficient network-assisted multicast and/or a flexible replication mechanism with support for subscriptions to keys or key ranges—which in itself may be in conflict with the promised consistency model.

5.7.2 Hybrid Push-Pull Dissemination

Both of these problems can be solved simultaneously with a hybrid push-pull approach. By distinguishing between objects with small and large value through a threshold size or an explicit write parameter, they can be treated differently. The suitable trade-offs can be selected separately.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

For *small objects*, a push-based approach is most efficient. Directly include the value in the replication updates that are propagated to all replicas for that key. The value is immediately available (low read latency) and stays available in case other nodes are unreachable (high fault tolerance).

For *large objects*, instead, a pull-based retrieval approach can be favored. Only use the data store itself for announcement and coordination, e.g., include a reference to the owning node and a content hash. Should a client be interested in loading a blob, it can be retrieved with a separate bulk distribution mechanism. Since large objects generally are not high priority items, delayed availability and reduced fault tolerance is acceptable.

A dedicated configuration and firmware file deployment service could be implemented based on an efficient block exchange mechanism, e.g., hash lists as used in the *Syncthing Block Exchange Protocol* [Syn] or Merkle trees [Mer87]. An additional benefit of a separate service is that it allows for better retrieval decisions: By having more information available than the data store itself, only blocks that are required have to be transmitted, e.g., by checking whether the firmware version already matches the installed one or by only requesting blocks for changed partitions.

5.8 Networking Layer Improvements

Efficiency and security of the data store and its applications could be improved by tuning key aspects of the underlying networking stack. As these aspects are important to consider in a real-world deployment, we only mention them briefly as the scope of the work is primarily on the data store's design.

5.8.1 Efficient Routing

Without an efficient routing algorithm, multi-path networking topologies can cause much of the available link bandwidth to be wasted, transporting redundant messages the receiver has already seen. For better efficiency, a more sophisticated routing protocol can be used. For example, Babel is a distance-vector routing protocol which avoids loops and can adjust itself quickly to changes in the network topology, such as switching from a faulty link to an available redundant backup link [CS21].

5.8.2 Efficient Multicast

Multicast communication over unicast links is always possible by using application layer multicast, as we did in our prototype implementation for the peer replication messages. The sender can send a separate copy of the same message to every recipient. While simple to implement, the approach can limit scalability: It places a greater load on the sender/forwarding nodes and wastes bandwidth on the intermediate network links.

With support from the lower layers in the networking stack, a more efficient variant can be possible: network-assisted multicast. Copies of the message are only generated as

TU Bibliothek WIEN Your knowledge hub
Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

required by the forwarding nodes. Our replication messages are an ideal use case, as they are sent identically to every peer node. Large value object distribution for configuration or firmware updates, as described in Section 5.7, could also benefit.

5.8.3 Message Authentication

Secure authentication mechanisms are essential in any real-world deployment to ensure the integrity of received messages before trusting and processing their contents.

Otherwise, malicious adversaries can easily tamper with message content or inject entirely fabricated messages. Not only can they manipulate the value of any data store item, but also disrupt the operation of the entire DDS and compromise safety-relevant functionality of the system. By sending update operations with an arbitrary, far ahead causal context, for example, the FIFO stores in our NDC-based system are not allowed to merge them. Instead, they have to keep them in cache in case the "missed" messages arrive—which will never happen if they don't exist anywhere. These caches can easily be overwhelmed by lots of bogus operations, leading to high resource usage and even displacement of the authentic operations.

To protect from such attacks and increase the system's robustness, all nodes can calculate and attach a cryptographically secure digital signature to every sent message. The signature allows all participants to verify the authenticity of all received messages and ignore them otherwise. For use on constrained devices, the signing and verification algorithms have to be selected with care. Modern embedded platforms often already provide hardware acceleration for cryptographic operations. Additionally, the key material needs to be securely distributed to all legitimate nodes and kept up to date. To solve this problem well is a challenge in itself.

Research Question RQ–2

> What technologies and which software architecture can be used to provide that consistency model on constrained devices?

We find the NDC framework from Gonçalves et al. to be suitable as base technology for a distributed data store running on constrained devices. It provides eventual consistency with continuous replication and periodic anti-entropy synchronization for high availability. To support the consistency model demanded by RQ–1, we develop a priority-aware FIFO consistent view that can optionally be layered on top and is fully transparent to the client application. To facilitate testing and maintain modularity, we design our proof-of-concept prototype with clear, minimal interfaces towards the client application and towards the underlying network functionality. We make an effort to keep communication logic, consistency-specific processing, configuration context and background tasks separate from each other for unit testability and future extensibility.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

CHAPTER 6

Evaluation

We designed and prototyped a distributed key-value data store. In this chapter, we evaluate the design against the requirements defined in Chapter 4. Chapter 5 documents the details of the design process and the resulting prototype.

6.1 Qualitative Analysis

The prototype data store we developed is a proof-of-concept on how the functional requirements can be met. Because the implementation is not optimized well for performance and low resource usage, measuring these properties does not yield meaningful results. Instead, we argue based on how a production-grade application based on our design could meet the non-functional requirements.

6.1.1 Functional Requirements

FR–1: Limited Transmission Time

> Changed data items must be readable on *all reachable nodes* within *20 seconds*.

As the data store has to rely on the underlying networking layers to deliver its replication messages in time, by itself, it cannot ensure fulfillment of this requirement. However, the design makes an effort to keep the replication latency low. Due to the asynchronous replication of value updates, unreachable nodes do not delay or prevent messages from being sent out to other nodes (Section 5.2). Assuming the network delivers messages in time, the remaining sources of latency consist of the message processing time and the visibility delay introduced by the consistency model.

Under normal operation, the processing time on the receiving nodes is negligible. Using eventual consistency, changed values are readable as soon as the messages have been

processed. Using the FIFO consistency mode, however, visibility may be delayed until either the underlying network delivers all preceding replication messages (of equal or higher priority), or until the anti-entropy background process can retrieve them from a peer node (Section 5.3.2). The latter can easily exceed the mandated transmission time, but as a fallback mechanism, it is only relevant in case the network already has failed to deliver the replication messages in time.

FR–2: Unreachable Node Detection

> Unreachable nodes must be detected by *at least one* other node within *20 seconds*.

In Section 5.5, we considered possible variants of detecting faulty links or crashed nodes. Application-level, end-to-end heartbeats can utilize the infrastructure of the distributed data store while staying independent of its inner workings. Every node can confirm its liveness by regularly updating a well-known key in the data store (e.g., every five seconds). A detection mechanism can trigger after two or three missed heartbeat intervals. Additionally, since the FIFO consistency mechanism must preserve the visibility order, an up-to-date heartbeat signal queried in FIFO mode confirms that all values written by that node up until that point must have been received, too.

FR–3: Continuous Data Replication

> Data items must be continuously replicated across nodes.

The requirement for replication is met by the NDC-based design: The NDC-based data store uses a replication mechanism to broadcast changed items values and a robust anti-entropy background synchronization task to ensure any deviations do not persist (see Section 5.2.3 on page 68).

FR–4: High Availability

> The remaining nodes of the distributed data store must remain available for operations in case of a component failure (network partition or node crash).

The requirement for high availability is also met by the NDC-based design: Due to continuous replication between all nodes, read operations only need to query the nodes' local store, which always succeeds with low latency. Write operations can also always successfully update the local store. The replication mechanism is fully asynchronous: Write operations do not wait for delivery or confirmation of the replication messages.

To favor availability and low latency, our prototype design has two key differences from the original NDC framework (see also Section 5.2.1). First, we replicate keys to all nodes instead of only a selected subset. The data is always available locally, in contrast to the

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

original framework where the queried *coordinator* node might need to forward the query and wait for the result. The second key difference is that we do not expose the *quorum* parameter and default it to 1. With a quorum larger than 1, a read operation would need to wait for receiving the values from multiple replica nodes [Gon+15; Gon+17].

FR–5: Fault Recovery

When a node becomes reachable again after a component failure (network partition or node crash), changed data items must be synchronized without any loss of data.

The requirement for fault recovery, too, is met by the NDC-based design. Order, lateness or duplication of replication messages do not affect the resulting value of a data item—the causal context always ensures that any merge operation only overwrites obsolete value versions (Section 5.3.1). Should a message get lost entirely, the periodic anti-entropy background synchronization process retrieves the missing data on its next run with any peer that received the update.

Should multiple nodes change the same items concurrently, *all* values are kept and returned after the merge. Due to NR–6 (No Multi-Writer Data Items) we do not expect this to happen in regular FDAS operation, but we have made an effort to keep the functionality intact, even for the FIFO consistency models. Otherwise, the data store would need an intrinsic method for conflict resolution instead of simply leaving the task to the application itself.

FR–6: FIFO/PRAM Consistency

Data item changes issued by a specific node must be observed in the order they were issued in, by every node in the system.

The requirement for data consistency is met by the FIFO extension to the NDC framework (Section 5.3.2 on page 70). When requested by a client session, changes issued by the same node (and, in turn, all its clients) will be observed strictly in the order they were executed in. Changes issued by different nodes can appear in any order.

However, a client might not be able to observe every *single* value version as multiple versions can appear at the same time. Even when using multiple sessions concurrently, a client's view will be consistent as long as two conditions are met: The sessions must not use different consistency modes and the client must not switch between different nodes (*sticky connection*). Otherwise, the obtained query results are no better than eventual consistency and must be handled with the appropriate caution.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

FR–7: Priority for Important Items

Changes to important data items should be replicated with priority. In case of network congestion, low priority items should be dropped first. With a priority-aware FIFO consistency model, missing updates of low-priority items should not delay delivery of high-priority data items (*priority inversion*).

Similar to FR–1, this requirement cannot be delivered by the data store alone. The store has to rely on the underlying network to do priority-aware congestion control. The priority-specific FIFO consistency stores, however, provide a useful middle ground between the comfort of the general FIFO consistency model and the low-latency of eventual consistency. By filtering and ignoring low-priority items, high-priority items can be shown earlier (Section 5.3.3 on page 77).

No.	Summary	Met through
FR–1	Limited Transmission Time	Low-latency design of the replication mechanism and the underlying network
FR–2	Unreachable Node Detection	Application level heartbeat mechanism
FR–3	Continuous Data Replication	NDC-based replication mechanism
FR–4	High Availability	NDC-based replication between all nodes
FR–5	Fault Recovery	NDC-based anti-entropy mechanism
FR–6	FIFO/PRAM Consistency	FIFO consistency mode
FR–7	Priority for Important Items	Priority-aware FIFO consistency mode and congestion control of the underlying network

Table 6.1: Summary of functional requirements

6.1.2 Non-Functional Requirements

NFR–1: Functional Scalability

The data store should be able to handle a growing number of participating replica nodes and stored data items without disproportionately impairing the expected functionality.

The five important data structures of the NDC framework consist of maps and sets on item keys and node/update ID pairs (*dots*, see also Section 5.6). In our prototype, the keys consist of text strings and the node and update IDs consist of 32-bit integer values. The data structures are implemented as hash maps and hash sets to optimize for an efficient lookup, which is particularly important during the anti-entropy synchronization process. That way, the time complexity for processing client operations and replication messages is $O(1)$ on average, with respect to the number of nodes and of stored keys.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Some stored metadata requires $O(n^2)$ (WM) or $O(n * m)$ (causal context per ST object) of memory space for n nodes and m keys, respectively.

In general, the data structures in the FIFO specific storage share these characteristics. Only when messages are lost or arrive out-of-order, partial segments must be additionally maintained (inserted, merged and retrieved) in a buffer—the complexity of these operations scales with the number of contiguous segments, but is also $O(1)$ on average with respect to the number of nodes and of stored keys. The FIFO implementation requires additional memory for a separate copy of the five NDC maps and sets per supported priority level, but the contained instances of keys, dots, priority vectors and values are immutable and can be shared between different consistency stores.

NFR–2: Fault Scalability

> The data store should be able to tolerate a growing amount of component faults without disproportionately impairing the expected functionality.

Faults can result in network partitions or node crashes. When a node is reachable again after a network partition, the anti-entropy synchronization can retrieve the missed updates all at once. When a node comes back up again after a crash, the same is true. Only updates which were neither sent to any other node nor saved in durable storage are not recoverable. This may be a problem when the node counter was saved to durable storage, but the message itself was not, and the node creates a gap in its assigned message IDs after recovery—as the message does not exist anywhere, the gap can never be filled. An implementation with durable storage must take care of such corner cases, but we did not implement a solution (see also Section 5.6.1).

When messages are lost or extensive reordering is required, the FIFO mechanism has to maintain (insert, merge and retrieve) pending segments (see also Section 5.3.3). The time complexity of buffer inserts and lookups grows with the number of contiguous segments instead of the number of individual missing updates—a network partition might cause 100 missing messages, but if their message IDs fall into a contiguous range, only one partial segment is needed. The size of the anti-entropy synchronization response grows with the number of keys with missing updates and the number of FIFO priority levels. For the FIFO specific storage instances, buffered partial segments may also be included. The number of segments to be transferred grows with the number of affected keys and the number of missed consecutive updates.

Non-reachable nodes can inhibit the eviction of metadata for obsolete value versions (from the DKM and WM). To prevent exhaustion of storage space, some cleanup strategy is recommendable for nodes that never return. Due to NR–4, we did not design a mechanism with our prototype.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

NFR–3: Modularity

> Different concerns of the data store should be kept separate in the software architecture. Different components should be modular, encapsulate their internal details and have well-defined interfaces with minimal surface area.

In our prototype, we separate different layers and components with abstract interfaces to avoid unintended coupling and enable module replacement with little effort. For example, we defined the client-to-node application programming interface (API) and the node-to-node communications interface first and wrote unit tests against simple in-process stub implementations. Only later, once the algorithms were functioning reasonably well, we swapped them against a networking implementation capable of connecting different processes and machines. By using the Protocol Buffers format for message serialization and the ZeroMQ protocol as communication middleware, each can easily be swapped out against similar implementations. Similarly, the different consistency-model-specific data store instances hide their internal structure and expose only a limited set of non-destructive methods to other parts of the program. Apart from keeping the implementation details malleable, this design also simplifies refactoring and can help prevent programming errors.

NFR–4: Low Resource Usage

> A node has to run well on resource constrained embedded devices.

The NDC-based design has the advantage that it requires only simple calculations from the participating nodes and has low metadata overhead. The metadata for obsolete value versions can be removed as soon as every node has confirmed replication (using the WM cache of node clocks). In their paper, Gonçalves et al. evaluate their data store against a similar implementation using Merkle trees. They find that the NDC variant, in comparison, requires less network bandwidth and provides lower replication latency than the Merkle tree variant in scenarios where there is little node churn and the changes have little spatial locality [Gon+17].

No.	Summary	Met through
NFR–1	Functional Scalability	Efficient value lookups
NFR–2	Fault Scalability	Efficient anti-entropy synchronization and segment buffering
NFR–3	Modularity	Modular design
NFR–4	Low Resource Usage	NDC-based replication

Table 6.2: Summary of non-functional requirements

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

6.2 Testing

To analyze our prototype implementation, we utilized specialized testing software for distributed systems. We are interested in the implementation's behavior in the presence of simulated stress conditions. These tools do not prove correctness, but they can reveal flaws in the design, in our prototype implementation and even in our understanding of the problems that occur in distributed systems.

6.2.1 Maelstrom and Jepsen

We use the Maelstrom workbench [Kinb], based on the Jepsen testing library [KPa], both by Kingsbury. These tools are written in the Clojure programming language, a Lisp dialect for the Java Virtual Machine. Maelstrom runs random data operations generated by the *generator* against the implementation and records a log of all issued operations. These include the operation parameters, the obtained result or error, and the invocation and completion timestamps. After the run, the history of the operations can be analyzed by a workload-specific set of *checkers*. These checkers can report problems such as consistency violations or generate additional results, e.g., render a graphical timeline or calculate message statistics. Any errors logged by the tested nodes are also included.

Each test run is highly configurable: The *workload* specifies the semantics of the distributed system, i.e., which RPC functions are available for the distributed system under test and what generator and set of checkers to use. Apart from the workload, the number of node instances to spawn must also be set. Several other parameters are available to fine tune the aggressiveness of the test run. These parameters include the amount of workers that issue operations concurrently, the approximate target rate of requests per second, simulated network latency and message loss. For the latter, *nemesis* processes can be enabled to inject random faults.

JSON Protocol

As a prerequisite for testing, we had to swap out the networking layer of our proof-of-concept prototype implementation. We built an adapter for the Maelstrom JSON RPC protocol [Kinb, `doc/protocol.md`] over standard input/output instead of the Protobuf/ZeroMQ networking used up to that point. The client API needs to be able to handle requests sent by the simulated Maelstrom clients.

Swapping out the network layer should not affect the significance of the results: The same logical messages are just encoded differently and transferred over another asynchronous channel. The only effect we expect is a higher processing latency, since the encoding process and intermediate buffering may not be as efficient. However, it should be negligible compared to the latency of any real network connection and the simulated latency we plan to introduce.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

The internal messages used for node-to-node replication and anti-entropy synchronization, too, need to be wrapped in a JSON envelope for routing. Routing via Maelstrom provides two benefits: Our internal messages will also be affected by the simulated latency and injected faults. And Maelstrom will render them in the generated graphical timeline and message statistics.

The protocol includes a list of error codes hat can be returned for a non-succeeded operation. Maelstrom assigns them two distinct classes: *definite* and *indefinite.* A definite error code (e.g., *abort* or *not-supported*) means that the operation definitely did not (and never will) happen. An indefinite error code (*timeout* or *crash*), instead, means that the operation might or might not have taken place. It could also still take effect at a later time. While operations resulting in a definite error can be safely ignored by a checker to improve performance, indefinite errors must still be considered.

6.2.2 Lin-kv Workload

Out of the predefined Maelstrom workloads [Kinb, `doc/workloads.md`], *lin-kv* is the best matching starting point for our scenario. It simulates and validates a linearizable, non-transactional key-value store with three available RPC operations: write, read, and atomic compare-and-swap (CAS). Internally, each key of the distributed data store is mapped to a model of a linearizable register. When the simulation run is complete, the recorded history is evaluated with the Knossos checker [KPb] for any consistency violations.

To get familiar with these tools and to debug our implementation of the JSON protocol, we tested our prototype against the *lin-kv* workload first. For CAS, we implemented only a non-atomic read-then-write compound operation for compatibility. Simply aborting and returning a not-supported result would have been sufficient, too, but resulting in worse availability results.

For concurrent write operations, our prototype does not do any conflict resolution. Instead, all concurrent values are returned as a list on the next read. Since the Maelstrom RPC result (*read_ok*) can only handle a single value, we abort the operation and return an error code, instead. Error results reduce the calculated availability, but are irrelevant for the consistency check.

Results (lin-kv)

We configured the workload to run with a cluster of four simulated nodes, 16 concurrent client processes and a duration of 60 seconds. The other simulation parameters were varied, as shown in Table 6.3. Note that since Jepsen randomizes the generated operations and injected faults, these are one time results and not exactly comparable and repeatable.

The first four columns show the test configuration: consistency mode, target rate of operations per second, mean simulated latency and simulated network faults. The

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Mode	Configuration Rate	Latency	Faults	Results R/W Rate	Linearizable?
Event.	5	0	None	1.0 / 1.7	OK
Event.	5	0	Partition	1.2 / 1.7	Not OK
Event.	5	250	None	1.0 / 1.7	Not OK
Event.	5	250	Partition	1.0 / 1.6	Not OK
Event.	500	0	None	113.0 / 157.1	Not OK
Event.	500	0	Partition	116.7 / 162.6	Not OK
Event.	500	250	None	35.0 / 111.3	Not OK
Event.	500	250	Partition	40.7 / 111.3	Not OK
FIFO	5	0	None	1.2 / 1.6	OK
FIFO	5	0	Partition	1.3 / 1.6	Not OK
FIFO	5	250	None	1.1 / 1.7	Not OK
FIFO	5	250	Partition	1.2 / 1.5	Not OK
FIFO	500	0	None	114.8 / 162.1	Not OK
FIFO	500	0	Partition	114.8 / 161.4	Not OK
FIFO	500	250	None	31.5 / 106.3	Not OK
FIFO	500	250	Partition	44.3 / 113.4	Not OK

Table 6.3: Consistency results for *lin-kv* runs (4 Nodes, 60s, No Read Quorum)

Mode	Configuration Rate	Latency	Faults	Results R/W Rate	Linearizable?
Event.	5	0	None	1.0 / 1.6	OK
Event.	5	0	Partition	0.7 / 1.1	OK
Event.	5	250	None	1.2 / 1.6	Not OK
Event.	5	250	Partition	0.6 / 1.1	Not OK
Event.	500	0	None	114.1 / 156.0	Not OK
Event.	500	0	Partition	45.0 / 65.8	Not OK
Event.	500	250	None	2.1 / 3.4	Not OK
Event.	500	250	Partition	0.9 / 1.8	Not OK
FIFO	5	0	None	1.2 / 1.8	OK
FIFO	5	0	Partition	1.3 / 1.7	OK
FIFO	5	250	None	0.9 / 1.6	Not OK
FIFO	5	250	Partition	0.4 / 0.9	OK
FIFO	500	0	None	109.9 / 158.5	Not OK
FIFO	500	0	Partition	28.4 / 43.2	Not OK
FIFO	500	250	None	2.1 / 3.6	Not OK
FIFO	500	250	Partition	0.7 / 1.8	Not OK

Table 6.4: Consistency results for *lin-kv* runs (4 Nodes, 60s, Read Quorum 4)

remaining columns show the measured results: the rate of successfully completed read and write operations per second and the result of the linearizability checker.

We observe that with a low mean target rate and without network partitions, a run may result in a linearizable history. With a higher mean target rate and/or a partition nemesis that isolates network segments randomly from each other, however, consistency errors are found with a high probability.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

With the same settings, but a full read quorum of 4 nodes, more runs with a low mean target rate are linearizable Table 6.4. The reason lies in the complete non-availability during periods of network partition: With no read operations returning values, none of them can be inconsistent. The mean rate of successfully completed read operations is significantly reduced by the presence of network partitions.

Identified Problems of the Prototype

This test allowed us to fix minor problems not noticed up until that point. The node's internal data store could stay locked and block other operations while waiting for a reply during the anti-entropy synchronization or fetching a value with a read quorum. Some internal RPC commands did not reliably cancel after the configured timeout, leaving the node blocked and unable to process further requests.[1]

When fetching a value with a read quorum $1 < RQ < N$, no further nodes were queried in case a node of the randomly chosen initial set was not responsive. The read operation resulted in a timeout instead. Additionally, fetching a value with a read quorum is only compatible with our implementation of FIFO consistency when the queried replicas do not vary. Otherwise, the resulting inconsistent view is comparable to a non-sticky client switching between different nodes. Since support for read quora is not among our requirements and not well tested, these problems were not unexpected.

6.2.3 Fifo-kv Workload

For more meaningful results, we have to develop a better matching Maelstrom workload for our implementation. *fifo-kv* can reuse the existing read/write RPC operations of *lin-kv*, but needs a modified generator and a different checker.

Generator

The existing generator searches through every permutation of concurrent operations to reveal any dormant linearizability problems. For every key to test, randomly mixed read/write operations are generated with random values. These are then invoked on a random checker process that happens to be free. To match the goals of our design, we need to reduce that entropy step-by-step.

First, we remove the non-supported CAS operation and modify the generation of *write* operations: Instead of using random numbers, we use strictly increasing values that are only assigned once. This way, values returned by later *read* operations can be traced to the associated write unambiguously, which is a prerequisite for the FIFO/PRAM checking algorithm [Wei+16].

In addition, concurrent writes from different nodes need to be restricted. We only want to allow writes to keys "owned" by a specific client/node (due to Non-Requirement NR–6).

[1]These problems are typical examples of the challenges described in Section 2.2: While the network works sufficiently well, they can lay dormant for a long time, but cause catastrophic results later.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Due to the design of Jepsen, that is not straightforward to accomplish. At the time of generation, it is not yet known which process will be chosen to execute the write. At the time of execution, however, it is already too late: We can change the key or the target node of the operation, but the change will not be reflected in the already stored history. One simple solution is to filter the operations on execution: We can immediately abort any operation that would write to a key owned by a different node. Apart from the inefficiency, a drawback of this workaround is that the rate of operations per second and the availability statistics are not accurately calculated anymore.

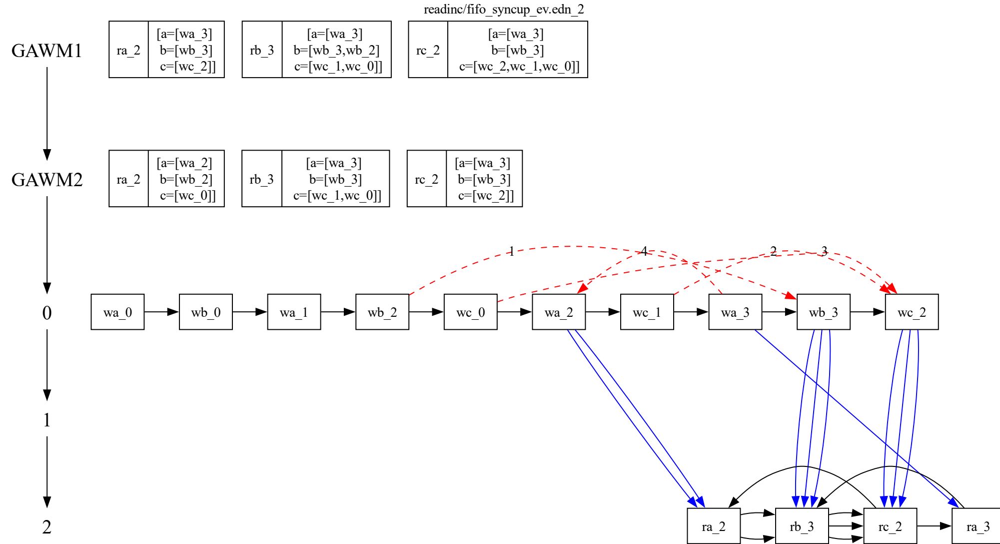
FIFO/PRAM Checker

To inspect the written history for inconsistencies, we cannot utilize the Knossos checker because it is too strict. Instead, we employ the checking algorithm described in Section 3.1.1 on page 32 [Wei+16]. Wei has kindly shared a Java implementation [Wei13] which we could use. We only had to implement an observation file loader for reading the Extensible Data Notation (EDN) trace file written by Jepsen (*history.edn*).

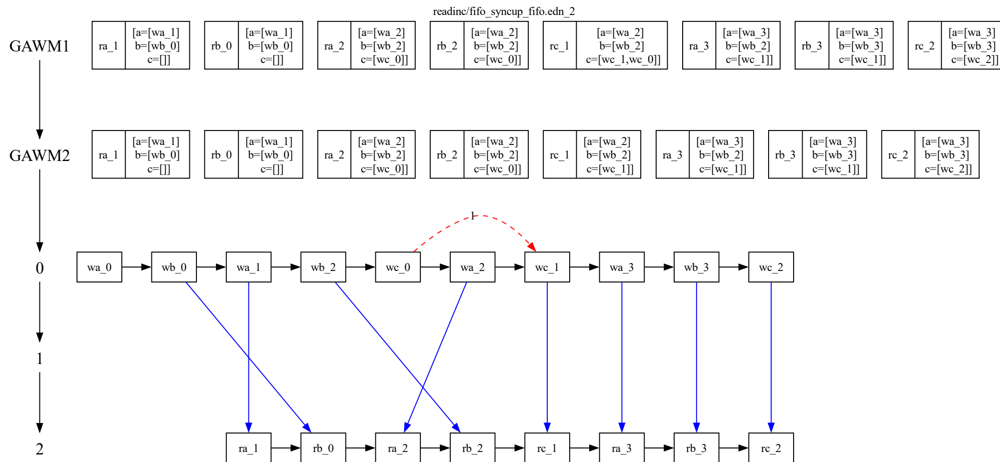
To verify our implementation, we extended our prototype's consistency unit tests to write out read/write traces, both in Eventual and in FIFO consistency mode sessions. While read operations only create entries in the mode-specific trace, write operations need to be included in both—they affect all (later) reads. These traces include the scenarios shown in Figure 5.5 and Figure 5.6 (pages 74/76). For illustration, we included the two graphs generated from checking the first scenario: In eventual mode (Figure 6.1a), the algorithm determines that the graph contains cycles—indicating an inconsistent order of operations was found—and terminates early. In FIFO mode (Figure 6.1b), in contrast, no cycles are found. All operations seem to be in FIFO-consistent order and the algorithm only terminates once it has processed the whole trace.

One problem only became apparent once we tried to check a large trace including simulated network faults. The checker would crash, being unable to find the corresponding write operation for a read value. The reason turned out to be quite simple: The write operation had indeed taken effect, but with the returned *write_ok* confirmation message being lost, Maelstrom had logged the result as a *timeout* error instead. We had to modify the history loading code to include all write operations that might have taken effect—successfully completed ones, but operations resulting in an *indefinite* error, too.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub



(a) Eventual mode: Inconsistent trace for node C (graph contains cycles)



(b) FIFO mode: Consistent trace for node C (acyclic graph)

Figure 6.1: FIFO checking graphs for the scenario from Figure 5.5

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Mode	Configuration Rate	Configuration Latency	Faults	Results R/W Rate	Results Visibility Latency	FIFO?
Event.	5	0	None	1.6 / 0.4	2 / 6	OK
Event.	5	0	Partition	1.7 / 0.4	2 / 6	OK
Event.	5	250	None	1.6 / 0.5	6 / 260	OK
Event.	5	250	Partition	0.8 / 0.3	4 / 259	OK
Event.	500	0	None	309.9 / 37.9	1 / 3	OK
Event.	500	0	Partition	304.8 / 38.3	1 / 3	Not OK
Event.	500	250	None	212.6 / 23.2	34 / 287	OK
Event.	500	250	Partition	171.6 / 23.0	28 / 287	Not OK
Event.	1000	0	None	591.9 / 72.7	1 / 2	OK
Event.	1000	0	Partition	528.8 / 72.5	1 / 2	Not OK
Event.	1000	250	None	295.6 / 30.2	27 / 279	OK
Event.	1000	250	Partition	309.4 / 36.1	18 / 277	Not OK
FIFO	5	0	None	2.0 / 0.4	2 / 7	OK
FIFO	5	0	Partition	1.8 / 0.5	2 / 7	OK
FIFO	5	250	None	1.4 / 0.3	15 / 268	OK
FIFO	5	250	Partition	0.9 / 0.5	2 / 386	OK
FIFO	500	0	None	306.7 / 38.5	1 / 3	OK
FIFO	500	0	Partition	288.3 / 38.4	1 / 3	OK
FIFO	500	250	None	181.6 / 20.0	44 / 296	OK
FIFO	500	250	Partition	168.7 / 20.1	36 / 311	OK
FIFO	1000	0	None	587.1 / 70.6	1 / 3	OK
FIFO	1000	0	Partition	479.7 / 72.2	1 / 2	OK
FIFO	1000	250	None	340.8 / 35.4	25 / 277	OK
FIFO	1000	250	Partition	220.7 / 29.3	7 / 296	OK

Table 6.5: Consistency results for *fifo-kv* runs (4 Nodes, 60s)

Results (fifo-kv)

Table 6.5 shows the results from varying the same parameters, but using the dedicated *fifo-kv* workload. Note the additional column showing the measured median visibility latency of a written value in milliseconds, both on the writing node itself and as end-to-end latency on other nodes (which includes network latency).[2]

As shown by the results, client sessions in eventual consistency mode quickly suffer from FIFO consistency errors when network faults are injected. Sessions that use FIFO mode, in contrast, are able to maintain a consistent view for the client. The data consistency comes at the cost of a slightly increased value for the visibility latency. Interestingly, while the consistency checks for runs with rate 5 or 500 only took a few seconds at most, checking the rate 1000 runs can already take several minutes. It is possible that the checker's single-threaded implementation is not very efficient. But most likely, the computational complexity of the algorithm simply results in a steep increase in computation time for the tens of thousands of operations that are logged in the history.

[2]The measured latencies allow a comparison of the runs with each other, but general performance conclusions should not be drawn: The prototype is not optimized for performance, the values were not measured accurately and do include significant overhead.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Summary

We evaluated the design qualitatively against the requirements. We found that a data store module built on that design can deliver the mandated properties in principle. Requirements such as FR–1 or FR–7, however, cannot be ensured by the data store alone. They depend on the implementations of the underlying support modules (networking, durable storage). The non-functional requirements are hard to evaluate for a design where many implementation details are either left open or entirely out of scope. Fault Scalability (NFR–2), in particular, depends on not only one, but all components involved in a system to handle faults well.

In addition, we used simulation testing to measure the prototype's behavior. We had to implement a JSON protocol adapter to allow Maelstrom to communicate with our prototype. We prepared a dedicated *fifo-kv* workload for Maelstrom with a customized generator and checker. Instead of all nodes writing to all keys, our generator generates writes for the keys owned by a node. Instead of generating values randomly, a unique value is used for each generated write operation. This enabled us to use the Read-Centric algorithm for the VPC-MU problem (multiple variables, unique values) to check the recorded traces for compliance with FIFO/PRAM consistency. Finally, we could execute 24 simulation runs to compare different configurations using our workload. The simulation results show that sessions in FIFO mode complied to FIFO consistency under all tested configurations—at the cost of some increased visibility latency in comparison to eventual sessions. We also found that the complexity of the checking algorithm seems to quickly reach the limits of computational feasibility. Traces of larger size, e.g., for long-term tests or load tests, would require splitting them into separate parts that can be analyzed separately.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

CHAPTER 7

Conclusion

Fire Detection and Fire Alarm Systems have a safety-critical role and must operate reliably. Therefore, a decentralized architecture is used to cover large areas or multiple buildings: Autonomous control units are networked together to form a Distributed Control Unit. One way for sharing the state with each other—while staying tolerant to network or node faults—is to use a continuously replicating distributed data store with key-value semantics. However, there are trade-offs involved concerning data consistency, availability, and operations latency.

This work was focused on the design of a data store module fit for that purpose. Inspired by the CAP and PACELC theorems, we wanted to find out whether relaxed consistency models—which can allow for higher availability and lower latency—can be sufficient for correct and safe operation on constrained hardware. We gathered requirements from the product criteria defined by the European Standards series EN 54 and found that the FIFO consistency model should be adequate for the basic use cases (RQ–1). The model simplifies application development by preventing write operations performed by the same node from being observed out-of-order.

After reviewing existing work, we based the design of our system on the node-wide dot-based clocks framework—which provides eventual consistency using robust asynchronous schemes for replication and anti-entropy. We extended the framework by an optional algorithm to ensure FIFO consistent views for client applications. Item priorities can be taken into account to prevent priority inversion—missing lower-priority items delaying delivery of high-priority items (RQ–2). In parallel, we used an iterative approach to build a prototype implementation. This helped to identify and correct problems early: For instance, we observed that in FIFO mode, the anti-entropy synchronization could result in an unnecessarily high visibility latency. We resolved the problem by having peers exchange buffered segments, too.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Finally, we evaluated the design qualitatively against the requirements and used simulation testing to measure the prototype's behavior. The simulation results show that the data store behaved as expected under network partitions and communications latency. Sessions in FIFO mode complied to FIFO consistency under all tested configurations—they only displayed a slightly increased visibility latency in comparison to eventual sessions.

In PACELC [Aba12] terminology, our proposed design belongs to the *PA/EL* class. Instead of strict consistency, we prioritize availability during a network partition, and lower latency during normal operation. It may seem that an important quality is given up here. However, strict consistency can work against functional safety: Systems based on Linearizability or Sequential Consistency cannot stay available during some network failures—they have to pause operations. This includes reading already existing data from local storage: A system with strict consistency must ensure that the data has not gone stale, as another node could have written a new version in the meantime.

During the analysis, we found that the existing solution (Section 4.1) also provides FIFO consistency, but has the drawback that the replication process is synchronous and tightly coupled between nodes. The scheme can be resource efficient under normal operation, but minor faults have a risk of causing cascading failures. Its synchronous nature makes it difficult to contain faults and isolate high-priority functions from problems occurring in lower-priority functions. In other words: By using asynchronous replication instead, our design did not have to choose weaker consistency guarantees to improve fault tolerance and, potentially, availability and throughput. However, the proposed design is not a complete solution and depends on other modules for lower-level functionality. Solving the associated problems well is an entirely different effort.

Future Work

A more comprehensive evaluation could help form a better picture, including measuring the behavior in various load, fault, and recovery scenarios. A formal specification of the algorithms could offer additional insight and enable verification of the consistency and safety properties, e.g., with TLA+ [Lam99] or PlusCal [Lam09]. Implementing causal consistency could ensure safe usage of stateful logical operators in all cases. Dynamic node membership could simplify deployment and maintenance: As we have seen in Raft (described in Section 3.3.1), cluster reconfiguration requires careful consensus to rule out the risk of split-brain scenarios. Advanced strategies for data partitioning and distribution could improve scalability and reduce resource usage on nodes. And lastly, with the growing interconnectedness of systems, *safety* increasingly depends on *security*. Authentication and validation of received data are a prerequisite for guaranteeing the integrity of the data store. The trust model from peer-to-peer computing—where there is no single central authority and the peers do not trust each other—could be useful in an embedded context, too.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

APPENDIX A

NDC Framework Extensions

In this supplemental chapter, we include pseudocode for our extensions to the algorithms of the node-wide dot-based clocks (NDC) framework [Gon+17, p. 199] by Gonçalves et al.

A.1 Priority-Aware FIFO Algorithm

While the base design only provides eventual consistency to its clients, these extensions maintain a separate FIFO store for every supported minimum priority level. Any updates that are not yet FIFO compliant are kept in a separate segment buffer that is not visible to clients. For a general overview, see Section 5.3 on pages 69–81.

A.1.1 Data Structures

The NDC framework keeps five distinct data structures on a data store node i [Gon+17, p. 198]. In addition to this original set for eventual consistency, we keep a separate set for every minimal FIFO priority level p we want to be able to answer client queries for: Node Clock ($NC_{i,p}$), Dot-Key Map ($DKM_{i,p}$), Watermark ($WM_{i,p}$), Non-Stripped Keys ($NSK_{i,p}$) and Storage ($ST_{i,p}$). The contents of these data structures correspond to the *primary segments* that are always kept FIFO compliant.

In addition, we need storage for the buffered *partial segments*. For every node ID, $BS_{i,p}$ contains a separate set of segments. Each of these segment covers a non-overlapping range of contiguous update IDs and contains a mapping of *keys* to *objects*. It is similar to the structure of $ST_{i,p}$, but updates issued by different source nodes are kept strictly separate instead of being merged together. The reason is that the FIFO condition is evaluated by node ID and they may become compliant at different times. $BS_{i,p}$ is kept *normalized*, i.e., the segments are sorted for fast access and adjacent segments are merged together for efficient storage.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Using the notation from Gonçalves et al., the structure of the partial segments buffer can be described as:

$$BS \doteq \mathbb{I} \hookrightarrow \underbrace{(R \times (\mathbb{K} \hookrightarrow O))}_{\text{Partial Segments}}$$

With R being non-overlapping intervals of update IDs and O being the NDC objects, extended with a priority/distance map DV per value version:

$$O \doteq (\underbrace{(\mathbb{I} \times \mathbb{N})}_{\text{Dots}} \hookrightarrow (\mathbb{V} \times \underbrace{(\mathbb{P} \hookrightarrow \mathbb{N})}_{\text{DV}})) \times \underbrace{(\mathbb{I} \hookrightarrow \mathbb{N})}_{\text{CC}}$$

And the following symbols for sets:

- $\mathbb{I}$ . . . Node identifiers
- $\mathbb{N}$ . . . Update identifiers (natural numbers)
- $\mathbb{K}$ . . . Keys
- $\mathbb{V}$ . . . Values
- $\mathbb{P}$ . . . Priority levels

A.1.2 FIFO Hooks

When the data store receives an updated object from a client or replication operation, it is applied using the `update` function (Algorithm A.1). We rename the existing NDC function to `eventual_update` and wrap it to call the priority-specific FIFO `fifo_update` hook once for every priority level. The `fetch` object fetching function and the `sync_clock` and `sync_repair` procedures for anti-entropy synchronization are wrapped in a similar fashion to collect from or apply to either the eventual store or the priority-specific FIFO stores.

A.1.3 Replication

Update Processing

The `fifo_update` procedure considers and merges or buffers a newly received update (Algorithm A.2). In case the key's assigned priority level is below the minimum priority for the store, the update is not relevant and can safely be ignored. If allowed by the FIFO condition, i.e., merging would not create gaps in the node clock (`fifo_can_merge`), the object with all version values is merged immediately into the primary segment. This is the common case in the absence of dropped or reordered updates.

Otherwise, a partial segment is constructed for every separate version value contained in the object and added to the partial segment buffer using `fifo_buffer_segment`. For the lowest priority level, the range covers only the one update ID itself. For higher levels, it covers all update IDs since the predecessor of equal-or-higher priority. Afterwards, `fifo_check_segments` is called to check whether now, any of the partial segments from that node can be merged.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Wien Bibliothek Your knowledge hub

Algorithm A.1: Required Hooks at Node i

```
1  Function update(Key k, Object o):
2      o := eventual_update(k, o)
3      for p ∈ ℙ do fifo_update_p(k, o)
4      return o
5  Function fetch(Key k, Priority p):
6      if p = ∅ then
7          o := eventual_fetch(k)
8      else
9          o := fifo_fetch_p(k)
10     return o
11 Function sync_clock(SyncRequest req):
12     rep[∅] := eventual_sync_clock(req[∅])
13     for p ∈ ℙ do
14         rep[p] := fifo_sync_clock_p(req[p])
15     return rep
16 Procedure sync_repair(SyncResponse rep):
17     eventual_sync_repair(rep[∅])
18     for p ∈ ℙ do fifo_sync_repair_p(rep[p])
```

Algorithm A.2: FIFO Update Processing at Node i, Priority Store p

```
1  Procedure fifo_update(Key k, Object o):
       // ignore updates for lower-priority keys
2      if prio(k) ≤ p then return
       // short circuit the common case: try to merge straight in
3      if {(n, c, v, dv) ∈ o | fifo_can_merge(n, fifo_get_range(c, dv))} then
4          fifo_merge(k, o)
5          return
       // buffer a partial segment for each version value
6      for (n, c, v, dv) ∈ o do
7          o' := split(o, n, c)
8          segment := (fifo_get_range(c, dv), k, o')
9          fifo_buffer_segment(n, segment)
10         fifo_check_segments(n)
11 Function fifo_can_merge(NodeId n, Range r):
       // FIFO condition: no gaps in between node clock and range
12     return min(r) ≤ base(NC_{i,p}) + 1
13 Function fifo_get_range(UpdateId c, PriorityDistance dv):
       // range covers update ID and all since the predecessor
14     predecessor := c − dv[p]
15     return [predecessor + 1, c]
```


Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm A.3: FIFO Segment Operations at Node i, Priority Store p

```
1  Procedure fifo_buffer_segment(NodeId n, Segment s):
2      BS_{i,p}[n] := BS_{i,p}[n] ∪ s
       // sort and merge adjacent segments (implementation omitted)
3      normalize(BS_{i,p})
4  Procedure fifo_check_segments(NodeId n):
       // merge partial segments with no gap left
5      while fifo_can_merge(n, range(min(BS_{i,p}[n]))) do
6          (r, k, o) := pop(min(BS_{i,p}[n]))
7          fifo_merge(k, o)
8  Procedure fifo_merge(Key k, Object o):
       // add all update IDs from range to node clock
9      for (n, c, v, dv) ∈ o do
10         covered := fifo_get_range(c, dv)
11         NC_{i,p}[n] := NC_{i,p}[n] ∪ covered
       // merge into the primary segment
12     store(k, o)
```

Partial and Primary Segment Operations

Whenever a partial segment is added to the buffer within `fifo_buffer_segment`, the buffers contents are normalized. The segments are sorted by the first update ID of their range and adjacent segments are merged together. The ordering allows `fifo_check_segments` to efficiently query and test the first partial segment from the buffer. If no gaps remain between node clock and the first update ID of the range, it can be removed and merged into the primary segment using `fifo_update`. In contrast to the `update` function from the NDC framework, we first have to add all covered update IDs to the node clock. Otherwise, gaps would remain for the never-seen updates of lower priority levels and prevent the FIFO store from making progress.

A.1.4 Anti-Entropy

Sync Clock

The NDC *anti-entropy* algorithm runs as a periodic background task. The node A chooses a random peer node B to sync with and sends its node clock. B receives the node clock, and within the `sync_clock` function, computes the differences to its own clock to return any missing objects for node A. For the FIFO specific data store, we wrap this function as `fifo_sync_clock` and run it once for every priority level. We extend the response to include partial segments from the buffered segments cache $BS_{i,p}$. We include any segment whose range covers update IDs missing from the peer's node clock.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Algorithm A.4: FIFO Anti-Entropy at Node i, Priority Store p

```
1  Function fifo_sync_clock(NodeId j, NodeClock NC_{j,p}):
2      (i, NC_{i,p}, O) := sync_clock(j, NC_{j,p})
       // include partial FIFO segments missing from peer clock
3      for (n, s) ∈ BS_{i,p} do
4          if max(range(s)) > base(NC_{j,p}[n]) then
5              S := S ∪ {n, s}
6      return (i, NC_{i,p}, O, S)
7  Procedure fifo_sync_repair(NodeId j, NodeClock NC_{j,p}, KeyObjects O,
    PartialSegments S):
       // call NDC sync_repair to merge in the received objects
8      sync_repair(j, NC_{j,p}, O)
       // merge FIFO peer clock to close any gaps in our clock
9      for {n ∈ NC_{i,p} | n ∈ peers(i)} do
10         NC_{i,p}[n] := NC_{i,p}[n] ∪ NC_{j,p}[n]
       // add received partial segments to our buffer
11     for {(n, s) ∈ S | n ∈ peers(i)} do
12         fifo_buffer_segment(n, s)
13         fifo_check_segments(n)
```

Sync Repair

Upon receiving the response, node A calls the `sync_repair` procedure. We wrap this function as `fifo_sync_repair` and run it once for every priority level. First, we call the `sync_repair` procedure to store the received objects and update all data structures. In contrast to the original algorithm, we can then merge the node clock entries for all nodes instead of only for the peer node. Since the sending side is also a FIFO compliant data store, we can trust (and verify) that the entries contain no gaps. Additionally, we add the received partial segments to our buffer and check whether any of them are now gap-free and can be merged into the primary segment.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Overview of Generative AI Tools Used

"I hereby declare that, with the exception of the written text included in the appendix section, no generative AI tools were used in the creation of this thesis. Furthermore, no substantial text passages were adopted from any generative AI tools.

This declaration includes the following prompt and the application used to generate the aforementioned text in the appendix section:

Prompt: Write a short declaration for my thesis in which I declare that, with exception of this written text in the appendix section, no generative AI tools were used and no substantial text passages were adopted from such tools. Include this prompt and the application used, including the product name and version number/date.

Application: OpenAI's ChatGPT, based on the GPT-4 architecture, accessed in August 2024." ([Ope24])

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

List of Figures

2.1 The dependability tree 7
2.2 The fundamental chain of threats to dependability 8
2.3 Example topology of a networked FDAS 10
2.4 The Automation Pyramid 11
2.5 Communication artifacts in unreliable, asynchronous networks 16
2.6 Venn diagram illustrating the CAP theorem 21
2.7 Flowchart illustrating the PACELC decision tree 21
2.8 Relationship hierarchy between consistency models 23
2.9 Example timeline of a dirty read 29
2.10 Example timeline of a lost update 29

3.1 Examples of write sets that are causal cuts 35
3.2 Structure of a Merkle Search Tree 44
3.3 Bitmap Version Vector 46

5.1 Layered Architecture: Platform for the Application 64
5.2 Replicated Data Partitions 65
5.3 FIFO replication data at node B 71
5.4 Flowchart showing FIFO update processing 72
5.5 Lagging FIFO values without primary segment sync 74
5.6 Lagging FIFO values without synchronization of partial segments 76
5.7 FIFO Priority/Predecessor Distances 79
5.8 FIFO Priority/Predecessor Vector Representations 80
5.9 Partial FIFO segments per priority-specific store 80

6.1 FIFO checking graphs for the scenario from Figure 5.5 102

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

List of Tables

2.1 EN 54 standard series 6

3.1 Complexity classes for the VPC problem variants 33
3.2 Safety measures in Safety over EtherCAT 39

6.1 Summary of functional requirements 94
6.2 Summary of non-functional requirements 96
6.3 Consistency results for *lin-kv* runs (4 Nodes, 60s, No Read Quorum) . . . 99
6.4 Consistency results for *lin-kv* runs (4 Nodes, 60s, Read Quorum 4) 99
6.5 Consistency results for *fifo-kv* runs (4 Nodes, 60s) 103

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Acronyms

ACID atomicity, consistency, isolation, durability. 28

ACK Acknowledgment. 82

API application programming interface. 64, 66, 70, 77, 96, 97

BVV bitmap version vector. 45, 46, 67, 75, 82

CAS compare-and-swap. 17, 98, 100

CIE Control and Indicating Equipment. 6, 10, 11, 55, 63

CRC cyclic redundancy check. 38, 60

CRDT conflict-free replicated data type. 24, 25, 43, 44, 70

DAG directed, acyclic graph. 33

DCC dotted causal container. 46

DCU Distributed Control Unit. 1, 10, 11, 49, 54, 84, 105

DDS distributed data store. 1, 2, 17, 31, 56, 64, 65, 89, 105

DKM Dot-Key Map. 82, 84, 85, 95, 107

DS distributed system. 5

DST Daylight Saving Time. 16

DVV dotted version vector. 46, 47, 71, 73, 84

EDN Extensible Data Notation. 101

ERP Enterprise Resource Planning. 11

EtherCAT Ethernet for Control Automation Technology. 38

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

FACP Fire Alarm Control Panel. 6

FACU Fire Alarm Control Unit. 6, 54

FDAS Fire Detection and Fire Alarm System. 1–5, 7–10, 12, 49, 54, 61–63, 68, 83, 84, 93, 105

FIFO first in, first out. 27, 32, 36, 51, 53, 57, 58, 62, 66, 70–73, 75, 78–81, 84–86, 89, 92–95, 100, 101, 103–108, 110, 111

FSoE FailSafe over EtherCAT. 38

HAT Highly Available Transaction. 31

HMI Human Machine Interface. 12

ICS industrial control system. 12

IoT Internet of Things. 8

IP Internet Protocol. 37, 50

IPsec Internet Protocol Security. 60

IT information technology. 11

JSON JavaScript Object Notation. 97, 98, 104

KV key-value. 39, 43, 44, 46, 91, 98, 105

LAN local area network. 50

LSM log-structured merge tree. 42

MES Management Execution Systems. 11

MST Merkle Search Tree. 43–45

MTTR Mean-Time-To-Recovery. 39

MVCC multiversion concurrency control. 39

NC Node Clock. 71, 79, 82, 84, 85, 107

NDC node-wide dot-based clocks. 46, 67–73, 79, 81, 82, 84–86, 89, 92–96, 105, 107, 108, 110

NSK Non-Stripped Keys. 85, 107

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

NTP Network Time Protocol. 16

OCC optimistic concurrency control. 39

OT operational technology. 12

PLC programmable logic controller. 12, 63

PRAM Pipelined RAM. 27, 32, 33, 51, 57, 62, 70, 100, 104

QoS quality of service. 78

RDBMS relational database management system. 44

RPC remote procedure call. 68, 82, 97, 98, 100

SCADA Supervision, Control and Data Acquisition. 12

SIL Safety Integrity Level. 38

ST Storage. 70, 79, 85, 95, 107

TCP Transmission Control Protocol. 37, 50, 60, 65, 73, 78, 83

TLS Transport Layer Security. 60

UDP User Datagram Protocol. 60, 65, 78

VLAN virtual local area network. 78

VPC Verifying Pipelined-RAM Consistency. 32, 104

VPN virtual private network. 60

WM Watermark. 82, 84, 85, 95, 96, 107

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.

Bibliography

[Aba12] Daniel Abadi. "Consistency Tradeoffs in Modern Distributed Database System Design: CAP is Only Part of the Story". In: *Computer* 45.2 (Feb. 2012), pp. 37–42. DOI: 10.1109/mc.2012.33. URL: https://doi.org/10.1109/mc.2012.33.

[ALR01] Algirdas Avizienis, Jean-Claude Laprie, and Brian Randell. "Fundamental concepts of dependability". In: *Department of Computing Science Technical Report Series* (2001).

[AT19] Alex Auvolat and François Taïani. "Merkle search trees: Efficient state-based CRDTs in open networks". In: *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. IEEE. 2019, pp. 221–22109.

[Ath17a] Anish Athalye. *Porcupine: A fast linearizability checker in Go [Software]*. 2017. URL: https://github.com/anishathalye/porcupine (visited on 11/02/2023).

[Ath17b] Anish Athalye. *Testing Distributed Systems for Linearizability*. 2017. URL: https://anishathalye.com/testing-distributed-systems-for-linearizability/ (visited on 11/02/2023).

[Baia] Peter Bailis. *HAT, not CAP: Introducing Highly Available Transactions*. URL: http://www.bailis.org/blog/hat-not-cap-introducing-highly-available-transactions/ (visited on 10/19/2023).

[Baib] Peter Bailis. *Stickiness and Client-Server Session Guarantees*. URL: http://www.bailis.org/blog/stickiness-and-client-server-session-guarantees/ (visited on 08/31/2023).

[Bai+12] Peter Bailis et al. "The potential dangers of causal consistency and an explicit solution". In: *Proceedings of the Third ACM Symposium on Cloud Computing*. 2012, pp. 1–7.

[Bai+13a] Peter Bailis et al. "Bolt-on causal consistency". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. Association for Computing Machinery (ACM), June 2013. DOI: 10.1145/2463676.2465279. URL: https://doi.org/10.1145/2463676.2465279.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU Bibliothek WIEN Your knowledge hub

[Bai+13b] Peter Bailis et al. "Highly available transactions". In: *Proceedings of the VLDB Endowment* 7.3 (Nov. 2013), pp. 181–192. DOI: 10.14778/2732232.2732237. URL: https://doi.org/10.14778/2732232.2732237.

[BD13] Philip A. Bernstein and Sudipto Das. "Rethinking eventual consistency". In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. SIGMOD '13. New York, New York, USA: Association for Computing Machinery, 2013, pp. 923–928. ISBN: 9781450320375. DOI: 10.1145/2463676.2465339. URL: https://doi.org/10.1145/2463676.2465339.

[BEK14] Carsten Bormann, Mehmet Ersue, and Ari Keränen. *Terminology for Constrained-Node Networks*. RFC 7228. May 2014. DOI: 10.17487/RFC7228. URL: https://www.rfc-editor.org/info/rfc7228.

[Bre00] Eric A. Brewer. "Towards robust distributed systems (abstract)". In: *PODC: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*. ACM, July 2000. DOI: 10.1145/343477.343502. URL: https://doi.org/10.1145/343477.343502.

[CES71] E. G. Coffman, M. Elphick, and A. Shoshani. "System Deadlocks". In: *ACM Comput. Surv.* 3.2 (June 1971), pp. 67–78. ISSN: 0360-0300. DOI: 10.1145/356586.356588. URL: https://doi.org/10.1145/356586.356588.

[CS21] J. Chroboczek and D. Schinazi. *The Babel Routing Protocol*. RFC 8966. RFC Editor, Jan. 2021.

[EN 05a] EN 54–10:2002+A1:2005. *Fire detection and fire alarm systems—Part 10: Flame detectors—Point detectors*. 2005.

[EN 05b] EN 54–11:2001+A1:2005. *Fire detection and fire alarm systems—Part 11: Manual call point*. 2005.

[EN 06] EN 54–2:1997+A1:2006. *Fire detection and fire alarm systems—Part 2: Control and indicating equipment*. 2006.

[EN 08] EN 54–24:2008. *Fire detection and fire alarm systems—Part 24: Component of voice alarm systems—Loudspeakers*. 2008.

[EN 10] EN 54–23:2010. *Fire detection and fire alarm systems—Part 23: Fire alarm devices—Visual alarm devices*. 2010.

[EN 18a] EN 54–5:2017+A1:2018. *Fire detection and fire alarm systems—Part 5: Heat detectors—Point heat detectors*. 2018.

[EN 18b] EN 54–7:2018. *Fire detection and fire alarm systems—Part 7: Smoke detectors—Point smoke detectors using scattered light, transmitted light or ionization*. 2018.

[EN 19a] EN 54–13:2017+A1:2019. *Fire detection and fire alarm systems—Part 13: Compatibility and connectability assessment of system components*. 2019.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[EN 19b] EN 54–3:2014+A1:2019. *Fire detection and fire alarm systems—Part 3: Fire alarm devices—Sounders*. 2019.

[EN 21] EN 54–1:2021. *Fire detection and fire alarm systems—Part 1: Introduction*. 2021.

[Etc] *etcd: Raft library for maintaining a replicated state machine [Software]*. URL: https://github.com/etcd-io/raft (visited on 11/13/2023).

[FKK15] Thomas Frühwirth, Lukas Krammer, and Wolfgang Kastner. "Dependability Demands and State of the Art in the Internet of Things". In: *2015 IEEE 20th Conference on Emerging Technologies & Factory Automation (ETFA)*. 2015, pp. 1–4. DOI: 10.1109/ETFA.2015.7301592.

[GL02] Seth Gilbert and Nancy Lynch. "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services". In: *ACM SIGACT News* 33.2 (June 2002), pp. 51–59. DOI: 10.1145/564585.564601. URL: https://doi.org/10.1145/564585.564601.

[Gon+15] Ricardo Gonçalves et al. "Concise server-wide causality management for eventually consistent data stores". In: *Distributed Applications and Interoperable Systems: 15th IFIP WG 6.1 International Conference, DAIS 2015, Held as Part of the 10th International Federated Conference on Distributed Computing Techniques, DisCoTec 2015, Grenoble, France, June 2-4, 2015, Proceedings 15*. Springer. 2015, pp. 66–79.

[Gon16] Ricardo Jorge Tomé Gonçalves. *DottedDB: A Distributed Key-Value Store with "Server Wide Clocks" [Software]*. 2016. URL: https://github.com/ricardobcl/DottedDB (visited on 03/27/2024).

[Gon+17] Ricardo Jorge Tomé Gonçalves et al. "DottedDB: Anti-Entropy without Merkle Trees, Deletes without Tombstones". In: *2017 IEEE 36th Symposium on Reliable Distributed Systems (SRDS)*. 2017, pp. 194–203. DOI: 10.1109/SRDS.2017.28.

[Gro] EtherCAT Technology Group. *Safety over EtherCAT (FSoE) Introduction and Overview*. URL: https://www.ethercat.org/download/documents/Safety_over_EtherCAT_Overview.pdf (visited on 01/11/2024).

[Hev+04] Hevner et al. "Design Science in Information Systems Research". In: *MIS Quarterly* 28.1 (2004), p. 75. DOI: 10.2307/25148625. URL: https://doi.org/10.2307/25148625.

[HK15] Alex Horn and Daniel Kroening. "Faster linearizability checking via *P*-compositionality". In: (2015). arXiv: 1504.00204 [cs.DC].

[Jac88] V. Jacobson. "Congestion Avoidance and Control". In: *SIGCOMM Comput. Commun. Rev.* 18.4 (Aug. 1988), pp. 314–329. ISSN: 0146-4833. DOI: 10.1145/52325.52356. URL: https://doi.org/10.1145/52325.52356.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[KF09] Ingmar Kellner and Ludger Fiege. "Viewpoints in Complex Event Processing: Industrial Experience Report". In: *Proceedings of the Third ACM International Conference on Distributed Event-Based Systems*. DEBS '09. New York, NY, USA: Association for Computing Machinery, 2009. ISBN: 9781605586656. DOI: 10.1145/1619258.1619271. URL: https://doi.org/10.1145/1619258.1619271.

[Kina] K. Kingsbury. *Jepsen: Consistency Models*. URL: https://jepsen.io/consistency (visited on 07/24/2023).

[Kinb] Kyle Kingsbury. *Maelstrom: A workbench for writing toy implementations of distributed systems [Software]*. URL: https://github.com/jepsen-io/maelstrom (visited on 08/15/2024).

[Kle17] Martin Kleppmann. *Designing data-intensive applications*. Sebastopol, CA: O'Reilly Media, Mar. 2017.

[KPa] K. Kingsbury and K. Patella. *Jepsen: A framework for distributed systems verification, with fault injection [Software]*. URL: https://github.com/jepsen-io/jepsen (visited on 08/15/2024).

[KPb] Kyle Kingsbury and K. Patella. *Knossos: Verifies the linearizability of experimentally accessible histories [Software]*. URL: https://github.com/jepsen-io/knossos (visited on 08/15/2024).

[Lam09] Leslie Lamport. "The PlusCal Algorithm Language". In: *Theoretical Aspects of Computing-ICTAC 2009, Martin Leucker and Carroll Morgan editors. Lecture Notes in Computer Science, number 5684, 36-60.* (Jan. 2009). URL: https://www.microsoft.com/en-us/research/publication/pluscal-algorithm-language/.

[Lam78] Leslie Lamport. "Time, clocks, and the ordering of events in a distributed system". In: *Communications of the ACM* 21.7 (July 1978), pp. 558–565. DOI: 10.1145/359545.359563. URL: https://doi.org/10.1145/359545.359563.

[Lam98] Leslie Lamport. "The Part-Time Parliament". In: *ACM Trans. Comput. Syst.* 16.2 (May 1998), pp. 133–169. ISSN: 0734-2071. DOI: 10.1145/279227.279229. URL: https://doi.org/10.1145/279227.279229.

[Lam99] Leslie Lamport. "Specifying Concurrent Systems with TLA+". In: *Calculational System Design* (Apr. 1999), pp. 183–247. URL: https://www.microsoft.com/en-us/research/publication/specifying-concurrent-systems-tla/.

[Lin] *Linux Kernel (6.10.2)*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/tree/?h=v6.10.2 (visited on 07/20/2024).

[Low17] Gavin Lowe. "Testing for linearizability". In: *Concurrency and Computation: Practice and Experience* 29.4 (2017), e3928.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[LPS10] Mihai Letia, Nuno Preguiça, and Marc Shapiro. "Consistency without Concurrency Control in Large, Dynamic Systems". In: *SIGOPS Oper. Syst. Rev.* 44.2 (Apr. 2010), pp. 29–34. ISSN: 0163-5980. DOI: 10.1145/1773912.1773921. URL: https://doi.org/10.1145/1773912.1773921.

[LS88] Richard J. Lipton and Jonathan Sandberg. *PRAM: A scalable shared memory.* Tech. rep. CS-TR-180-88. Princeton University, Department of Computer Science, Sept. 1988.

[Mer87] Ralph C. Merkle. "A digital signature based on a conventional encryption function". In: *Conference on the theory and application of cryptographic techniques*. Springer. 1987, pp. 369–378.

[Net] *NetMQ: A 100% native C# port of the lightweight messaging library ZeroMQ [Software].* URL: https://github.com/zeromq/netmq (visited on 05/01/2024).

[Nie+18] Matthias Niedermaier et al. "Efficient Passive ICS Device Discovery and Identification by MAC Address Correlation". In: *Electronic Workshops in Computing*. BCS Learning & Development, Aug. 2018. DOI: 10.14236/ewic/ics2018.3. URL: https://doi.org/10.14236/ewic/ics2018.3.

[Nun] *NUnit: A unit-testing framework for all .NET languages [Software].* URL: https://nunit.org/ (visited on 06/15/2024).

[O'N+96] Patrick O'Neil et al. "The log-structured merge-tree (LSM-tree)". In: *Acta Informatica* 33.4 (June 1996), pp. 351–385. ISSN: 1432-0525. DOI: 10.1007/s002360050048. URL: http://dx.doi.org/10.1007/s002360050048.

[OO14] Diego Ongaro and John Ousterhout. "In Search of an Understandable Consensus Algorithm". In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, June 2014, pp. 305–319. ISBN: 978-1-931971-10-2. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro.

[Ope24] OpenAI. *ChatGPT*. 2024. URL: https://chatgpt.com/share/0bcc3401-b69c-4444-b919-1adb2a9e8181 (visited on 08/01/2024).

[Pet19] Alex Petrov. *Database Internals: A deep dive into how distributed data systems work*. O'Reilly Media, 2019.

[Proa] *protobuf-net: Protocol Buffers library for idiomatic .NET [Software].* URL: https://github.com/protobuf-net/protobuf-net (visited on 05/01/2024).

[Prob] *Protocol Buffers: Base 128 Varint Encoding.* URL: https://protobuf.dev/programming-guides/encoding/#varints (visited on 06/03/2024).

[SRC84] Jerome H Saltzer, David P Reed, and David D Clark. "End-to-end arguments in system design". In: *ACM Transactions on Computer Systems (TOCS)* 2.4 (1984), pp. 277–288.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub

[Sto12] Christian Storm. “Fault Tolerance in Distributed Computing”. In: *Specification and Analytical Evaluation of Heterogeneous Dynamic Quorum-Based Data Replication Schemes*. Wiesbaden: Vieweg+Teubner Verlag, 2012, pp. 13–79. ISBN: 978-3-8348-2381-6. DOI: 10.1007/978-3-8348-2381-6_2. URL: https://doi.org/10.1007/978-3-8348-2381-6_2.

[Syn] *Syncthing Block Exchange Protocol*. 2013. URL: https://docs.syncthing.net/specs/bep-v1.html (visited on 03/27/2024).

[Ter+94] D.B. Terry et al. “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.

[TS06] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems: Principles and Paradigms (2nd Edition)*. USA: Prentice-Hall, Inc., 2006. ISBN: 0132392275.

[TS99] Ulrich Tietze and Christoph Schenk. *Halbleiter-Schaltungstechnik*. 11th ed. Springer-Verlag, 1999.

[TW10] Andrew S. Tanenbaum and David J. Wetherall. *Computer Networks*. 5th ed. Upper Saddle River, NJ: Pearson, Sept. 2010. ISBN: 978-0-13-212695-3.

[Vog08] Werner Vogels. “Eventually Consistent: Building Reliable Distributed Systems at a Worldwide Scale Demands Trade-Offs Between Consistency and Availability.” In: *Queue* 6.6 (Oct. 2008), pp. 14–19. ISSN: 1542-7730. DOI: 10.1145/1466443.1466448. URL: https://doi.org/10.1145/1466443.1466448.

[VV15] Paolo Viotti and Marko Vukolic. “Consistency in Non-Transactional Distributed Storage Systems”. In: *CoRR* abs/1512.00168 (2015). arXiv: 1512.00168. URL: http://arxiv.org/abs/1512.00168.

[Wei13] Hengfeng Wei. *PRAM consistency checking in the context of distributed shared memory systems [Software]*. 2013. URL: https://github.com/hengxin/ConsistencyChecking (visited on 08/15/2024).

[Wei+16] Hengfeng Wei et al. “Verifying Pipelined-RAM Consistency over Read/Write Traces of Data Replicas”. In: *IEEE Transactions on Parallel and Distributed Systems* 27.5 (2016), pp. 1511–1523. DOI: 10.1109/TPDS.2015.2453985.

[Zho+21] Jingyu Zhou et al. “FoundationDB: A distributed unbundled transactional key value store”. In: *Proceedings of the 2021 International Conference on Management of Data*. 2021, pp. 2653–2666.

Die approbierte gedruckte Originalversion dieser Diplomarbeit ist an der TU Wien Bibliothek verfügbar
The approved original version of this thesis is available in print at TU Wien Bibliothek.
TU WIEN Bibliothek Your knowledge hub