



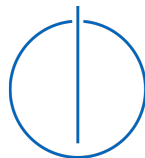
DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

**Shared Log with Persistent Memory and
eRPC**

Vincent Picking





DEPARTMENT OF INFORMATICS

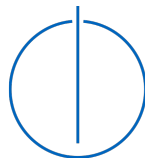
TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Shared Log with Persistent Memory and eRPC

Verteilter Log mit persistentem Speicher und eRPC

Author:	Vincent Picking
Supervisor:	Prof. Dr. Pramod Bhatotia
Advisor:	Dimitra Giantsidi
Submission Date:	15.11.2021



I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.11.2021



Vincent Picking

Acknowledgments

This thesis would not have been possible without the incredible support of my surroundings. First and foremost, I would like to thank my advisor Dimitra Giantsidi for her valuable advice and continuous support during the last months. Without her expertise, guidance, patience during our numerous meetings this thesis would not have been possible.

Further, I would like to thank Prof. Dr. Pramod Bhatotia who has put his trust in me and always had an open ear regardless of the problem.

A special thanks goes to Helma Schneider and Sophia Adelmeier for their continuously support throughout the years I have worked at the chair.

Most importantly, none of this would have been possible without the support of my family and friends over the past years. Especially, I want to express my gratitude to my parents Marion and Andreas for always having my back. Additionally, a special thanks goes to Matthias Linhuber, Sandesh Sharma, Paul Heidekrüger, Robert Jandow, Nikolai Madlener, Florian Angermeier, Daniel Bücheler and Marcel Ganß. Thank you for your valuable discussion, laughs and friendship over the years.

Lastly, I'm beyond grateful for having a wonderful partner at my side, Raffaella Böswald, who supported me anytime, anywhere, and kept me sane.

Abstract

Shared logs are one of the basic building blocks of distributed systems. They are used in system like Kafka [1], Amazon Aurora [2] or LogDevice [3]. The increased popularity of shared logs in the last decade is due to their offered simplicity and properties like Strong Consistency. They order events received from multiple clients, replicate them across multiple servers and make them accessible. Therefore, a shared log can drastically decrease complexity for the system built on top. As it is a core building block, the shared log needs to be as performant as possible. In recent years there have been advancements in the fields of networking, Persistent Memory, and multi-core processors. The open question is how these advancements can be leveraged in a shared log system. We designed and implemented *Ikaria*, a highly parallelized shared log with CRAQ as a replication algorithm. *Ikaria* makes use of asynchronous user-space networking and builds upon Persistent Memory, which offers byte-addressable access, persistent data storage, and performance close to volatile memory. Our results show that *Ikaria* offers high throughput for read-heavy workloads. Furthermore, we demonstrate that *Ikaria* scales well with an increasing number of servers, threads, and different log entry sizes compared to the original Chain Replication protocol.

Contents

Acknowledgments	iii
Abstract	iv
1 Introduction	1
2 Background	3
2.1 Persistent Memory	3
2.1.1 Persistent Memory Development Kit	4
2.1.2 Emulating Persistent Memory	4
2.2 Modern Networking	5
2.2.1 Remote Direct Memory Access	5
2.2.2 Data Plane Development Kit	6
2.2.3 Remote Procedure Calls	7
2.3 Replication Protocols	7
2.3.1 Consistency Models	7
2.3.2 Chain Replication	8
2.3.3 Chain Replication with Apportioned Queries	9
3 Design	10
3.1 Ikaria API	10
3.2 System Design	10
3.2.1 Network Layer	12
3.2.2 Replication Layer	13
3.2.3 Log Layer	19
3.3 Failure Recovery	22
4 Implementation	24
4.1 Network Layer	24
4.2 Replication Layer	25
4.3 Log Layer	26
4.3.1 PMDK	26
4.3.2 Log Abstraction	27

Contents

4.4	Bootstrap and Teardown	28
4.5	Multi-Threading	28
5	Evaluation	31
5.1	Experimental Setup	31
5.2	Chain Saturation	33
5.3	Operation Throughput	35
5.4	Tail Reading	39
5.5	Scalability	41
6	Related Work	44
7	Conclusion	47
7.1	Summary	47
7.2	Future Work	48
7.3	Threats to Validity	48
	List of Figures	50
	Bibliography	51

1 Introduction

Nowadays, most modern computer systems consist of several independently running servers that appear as a single service from the client's perspective: a distributed system [4, p. 2]. A distributed system can enable higher performance, scalability, and availability compared to a single server system. Unfortunately, this comes at the cost of increased complexity. The question arises of how to achieve a consistent state on all servers within the system. Distributed systems like ZooKeeper or databases implement individual solutions to generate a consensus between the different servers and manage their distributed state [5]. However, this results in complex systems which are challenging to instantiate and maintain.

A proposed solution for this problem is to build the system on top of a shared log abstraction. A log is one of the most basic data structures since it is only an array of entries consisting of several bytes. New entries are appended sequentially at the end of the log, so already written entries cannot be updated. When the log is shared, multiple clients can access it by reading and appending log entries [6]. Since the sequential log entry number totally orders the log entries, it defines its own logical time, where smaller entries occurred earlier than higher ones. Having such a logical time all servers can refer to is essential for a distributed system [7]. Therefore, a shared log solves the problem of ordering entries and replicating them across multiple machines. It can be used to solve the consensus problem, where different machines try to reach an agreement on certain values [6]. As a consequence, the log can be seen as a chain of decisions.

The given properties provided by the shared log make it possible to simplify the software running on top. Therefore, a shared log is at the core of available open-source systems like Kafka [1] or BookKeeper [8]. Another example would be Facebook which made an effort in the recent years to build Delos. This is a database which runs upon a shared log abstraction [5, 9]. Furthermore, Balakrishnan *et al.* [10] showed in their work that it is possible to implement existing systems like ZooKeeper [11] on top of a shared log with only one thousand lines of code. This is a lot less than the original implementation and drastically simplifies the system. Due to the rising popularity of the shared log abstractions, an increasing amount of shared log services like Google Pub/Sub [12] or AlibabaMQ [13] are offered by cloud providers [14].

A vital performance boundary for any shared log is that the entries must be persistent on a storage device. So in the case of a server failure, all entries are stored, and no entry is lost. Developers only had access to two types of storage for the last decades: volatile, fast memory, and solid, slow storage. Therefore, existing shared logs [6, 14, 15] are built on top of Solid-State-Drives (SSDs). SSDs provide higher performance than the older Hard-Disk-Drive (HDD), but they still suffer from a high I/O latency compared to memory. Through the technological advancements in the field of Non-Volatile Memory (NVM), mostly known as Persistent Memory (PM), there is now a third type of storage that offers performance close to memory, but persistent data storage. The first actual PM hardware module was released by Intel [16] in 2019. The question comes up for what kind of systems PM is relevant and could bring performance boosts. Besides the long-awaited release of Persistent Memory hardware, interesting advancements in the networking area have also been made. Namely, user-space networking is on the advance and promises fast packet processing [17, 18]. This thesis, therefore, investigates how to leverage these new advancements. To do so, two main research questions are examined in the following chapters:

- How to design a shared log with the recent advancements in networking and Persistent Memory?
- How to design a highly parallelized shared log that offers high throughput for read-heavy workloads?

Our contributions are the design and evaluation of *Ikaria*, a highly parallelized shared log system that makes use of modern networking and Persistent Memory, providing Strong Consistency.

This thesis is organized into seven further chapters. In chapter 2 we give a short overview of Persistent Memory, modern networking, and replication protocols. Afterward, we describe the design of our shared log *Ikaria* in chapter 3, with its API, single components, consistency models, and failure recovery. In chapter 4 the actual implementation of the shared log is presented and described in detail. We evaluate *Ikaria* in chapter 5, where we describe the experimental setup and present benchmark results. In chapter 6 other shared logs are presented and classified, CORFU, Scalog, Fuzzylog, Delos and Tango. We conclude the thesis in chapter 7 and open up possibilities for future work.

2 Background

This chapter describes hardware and software technologies that are referred to and used in this thesis. Furthermore, important theoretical concepts are introduced. Besides Persistent Memory and modern network technologies, we give a short overview of replication protocols, particularly chain replication and CRAQ.

2.1 Persistent Memory

Persistent Memory, or Non-Volatile Memory, is a new hardware class that can be classified between volatile memory and block-based storage (e.g. SSD, HDD). It provides persistent storage, byte-addressability, and latency which is close to Dynamic Random Access Memory (DRAM) while offering a lower cost-per-bit [19–21]. The byte-addressability opens up an advantage, especially when reading/writing small amounts of data. Compared to SSDs, for which commonly the smallest native read/write block is 4 kB, this can give performance and latency boosts [22, p. 6]. Similar to DRAM, PM is connected to the memory bus and therefore accessible for the CPU over its memory controller [23]. The only commercial available PM at the moment is the Intel Optane Persistent Memory module [16]. This Optane PM module can be used in two different operation modes [24, 25]. When using the *Memory Mode*, it is transparent for applications and can be used like standard volatile memory. Since this mode is not of particular interest for shared log applications, this thesis will focus on the second mode. In the *App Direct Mode*, the PM is accessed through a PM-aware file system (e.g. ext4, xfs, NTFS) [26] with the direct access extension (DAX) [23, 27] enabled. Usually, when accessing block-devices, the device pages are cached in the memory for faster access. For PM, this would include an unnecessary copy from DRAM to PM. Since PM offers a close to DRAM latency, the page cache should be avoided by enabling the DAX feature. Another advantage of circumventing the page cache is to eliminate the use of the kernel I/O system out of the data path [26]. There are two issues that have to be taken into consideration when working with PM hardware currently available, especially in the case of a system failure. First, the current hardware only supports atomic updates for values up to 8 B [28, p. 56]. Second, when cache lines are written back to the PM module, the correct write order is not guaranteed [23, p. 20]. When the writing process is interrupted by a system failure, it could lead to an inconsistent state.

2.1.1 Persistent Memory Development Kit

Intel introduced the Persistent Memory Development Kit (PMDK) in 2014, which is a collection of libraries and tools to help developers leverage PM [29]. The PMDK includes libraries for volatile and persistent use, but we are focusing on the persistent libraries in this thesis. These libraries automatically detect whether the underlying hardware is PM or another storage type and if the Operating System (OS) and CPU support PM. They use the correct persistence methods and make use of other platform depending optimizations (e.g. DAX, specific CPU operations) [23]. The *libpmem* library [30] is the basis for all persistent PMDK libraries. It provides low-level access to PM by abstracting away hardware tasks like cache flushing [31]. For this project especially important is the *libpmemlog* library [32] which offers functionality for managing a PM-resident log file namely *PMEMlogpool* is particularly relevant. This file consists of a header for metadata and is mapped continuously in the application's virtual address space [32].

2.1.2 Emulating Persistent Memory

Up until now, obtaining PM hardware has been a problem. Hence, finding the best way to do research and still have representative evaluations is a reoccurring question. There have been attempts to emulate PM e.g. by adding artificial memory access latency [33, 34] or by software emulations [35, 36]. As Yang *et al.* [25] show, there are several specific hardware characteristics of the Optane PM module which make emulating quite tricky. First, they confirm the Intel-provided bandwidth measurements [37]. They show that the module performs very differently when accessing sequentially or randomly. Intel states that for sequential/random read accesses the bandwidth is 7.6 GB/s and 2.4 GB/s, respectively, whereas for sequential/random write access the bandwidth is 2.3 GB/s and 0.5 GB/s, respectively [37, p. 350]. Second, the access granularity for the Optane PM module is 256 B [16], which means that smaller random accesses are less efficient. Furthermore, accesses which are a multiple of 256 B are more efficient. Like SSDs or DRAM, the PM module has a controller which performs tasks like bad-block management, wear-leveling, and write-buffering. Due to the internal controller's write buffer size of 16 kB, random writes which are having this access locality are performing better. Another difference to DRAM is that accessing the Optane PM with multiple concurrent threads leads to contention at the internal controller buffer and the memory controller buffer, which leads to a performance drop. These hardware characteristics, which Yang *et al.* [25] present in their work, have not been considered in the currently used emulator techniques. [25]

2.2 Modern Networking

Alongside the availability of low latency and high bandwidth PM, datacenter networking evolved in the past decades. Possible network data rates experienced a rise to up to 100 Gbit/s throughput nowadays and are assumed to reach 200 Gbit/s in the foreseeable future [19, 38]. For providing and leveraging these theoretical possible network speeds, there has been a focused development in specialized hardware-software co-designed technology like Remote Direct Memory Access (RDMA) [38, 39], Field Programmable Gate Arrays (FPGAs) [40], or programmable switches [41]. The traditional kernel-based TCP/IP network stack cannot provide the needed performance anymore for several reasons. Firstly, context switches between user and kernel space are too expensive [17, 18]. Secondly, there is a copy overhead since the received data must be copied from the kernel buffer to the application buffer [18]. Cai *et al.* [18] recently provided a detailed analysis of the network stack overheads. These problems are the reason why user-space networking is on the advance with, for example, the previously mentioned RDMA and the Data Plane Development Kit (DPDK) [42].

2.2.1 Remote Direct Memory Access

The general idea behind RDMA is to separate data movement overhead from data processing and to relieve the strain on the CPU. As the name indicates, RDMA enables a machine to manipulate data in another's machine DRAM over the network while bypassing the remote CPU [21]. RDMA provides lower latency and higher bandwidth by keeping the connection state and coordination between the two (or more) machines in the Network Interface Card (NIC). Due to this fact, RDMA can not only bypass the remote CPU, but there is also no need for an application to use the OS kernel-based network stack [21]. Since the connection state is kept in the NIC, special hardware called RDMA network interface cards (RNICs) is required. At the moment, RDMA is natively supported by fabrics like Infiniband or Cray Aries, which are commonly used in high-performance data centers [43]. For the ethernet fabric, there is RDMA over converged Ethernet (RoCE), which implements RDMA in existing Ethernet-based data centers by wrapping an Infiniband packet in an ethernet packet [39]. There are two classes of RDMA-Operations: one-sided and two-sided operations. When using one-sided operations like *read/write*, the client process is handling everything and manipulates the remote memory without the remote CPU involved. In two-sided operations namely *send/receive* the CPUs of both machines are involved [43]. When performing an *RDMA-write* request, the local RNIC copies the data to the remote RNIC buffer. From there, the remote RNIC's Direct Memory Access (DMA) engine will write the data over the PCI bus to the L3 cache bus or directly to the memory controller,

depending on the system [20]. Since RDMA has been designed for volatile memory in the first place, several challenges arise, especially with data durability, when using it with PM [19]. One problem occurs when data is sent to another machine, it is not guaranteed that the data is persisted once the *RDMA-write* completes [19]. It could still be in one of the buffers (e.g. RNIC-DMA engine buffer, L3 cache etc.) in the data transport chain on the remote host. Proposed solutions for this problem are sending a read-after-write request or performing an explicit cache flush when the data is in the L3 cache. These approaches sacrifice latency and performance [19].

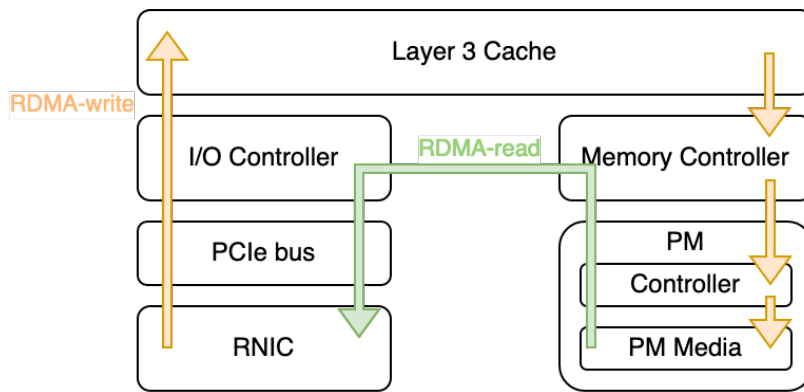


Figure 2.1: RDMA read/write data flow. This figure is inspired by the Intel RDMA PM documentation [44].

2.2.2 Data Plane Development Kit

The DPDK is an open-source project started by Intel that provides several data plane libraries and user-space poll-mode network drivers (PMDs). When the NIC receives new data, it does not send an interrupt to inform the CPU as in the currently used kernel network stack. Instead of this, it just copies the data directly via DMA into the memory [42]. The PMDs must now poll to get the received data which is more efficient than the interrupt-based approach [18, 42]. DPDK implements efficient data structures to fulfill tasks that were usually done by the kernel network, like a zero-copy and lock-free *Rx/Tx* ring buffer or hugepage-backed memory pools. Therefore DPDK enables to perform the whole network processing asynchronously and efficiently in the user-space [42, 45, 46].

2.2.3 Remote Procedure Calls

With a Remote Procedure Call (RPC), a local process can trigger a procedure on a remote host. Since this procedure may include any control flow, it is much more versatile than a single RDMA request [43]. As an asynchronous general-purpose RPC library, eRPC [47] tries to close the rising gap between sacrificing performance and generality. It is a polling-based end-to-end solution that builds upon existing transport layer protocols like RDMA, DPDK, or RoCE. Because these protocols do not offer reliable packet I/O, eRPC comes with packet loss handling and congestion control. Latencies are comparable to native RDMA. For example, an eRPC call has a median latency of 2.1 μs compared to 1.9 μs with an *RDMA-read* on an Infiniband fabric [41]. When transferring large messages, eRPC sends with a bandwidth of up to 75 Gbit/s. Every application thread creates an *RPC* object with a unique id, with which they can establish one-to-one connections between each other and *enqueue/receive* requests. For each request, a request and response hugepage-backed DMA-capable memory buffer, called *MsgBuffer*, has to be allocated in which the data is written. These two buffers belong to one request and cannot be reused in another request [41].

2.3 Replication Protocols

The high availability and performance of online services are critical requirements for most companies. Since a single server can fail unexpectedly, data or full services are replicated to stay available even when one server fails. After replication of the data, every server can answer queries for this object in some replication protocols, increasing the service's overall performance. Gavrielatos *et al.* [48] differentiate replication protocols whether they work in a centralized or decentralized manner. Centralized protocols incorporate a dedicated leader node that serializes requests like in chain replication (CR) [49]. For protocols working in a decentralized manner, all nodes need to reach an agreement over the total order. Further, Gavrielatos *et al.* subdivide protocols if they imply a total order over all requests for all keys, or in our case, multiple logs, (e.g. Raft [50]) or only imply an order over the requests per-key, or a single log, like CR [49] [48].

2.3.1 Consistency Models

A consistency model specifies which guarantees and assumptions a system offers when a user interacts with it through requests [51, p. 2]. It is essential to know for a user if read data could be stale or is guaranteed to be up to date. Various consistency models exist, which can be ranked by how strict the provided guarantees are. Strong Consistency, namely Linearizability or Sequential Consistency, hereby means that the

replication system is transparent for an accessing application, so it cannot differentiate between one server and a replicated multi-server system [51, p. 4]. All requests have to be executed in a total order consistent with the clients' request orders. Linearizability is even more strict than Sequential Consistency by demanding that a request must take effect before it completes [48]. Due to this offered transparency, even if Strong Consistency is not needed explicitly, a storage layer that provides this consistency level enables overlying layers a more straightforward way of working with the storage [49]. Implied by the CAP theorem [52] a stricter consistency model could mean a higher latency for finishing a request, performance penalties and under certain system failures lower availability guarantees [53] [51, p. 10]. For systems that prioritize high availability or low latency, a weak consistency guarantee that does not assure returning the most up-to-date value is sufficient. Eventual Consistency is a weak consistency guarantee, but since it provides low latency and availability even in the case of network partitions, applications are leveraging this approach (e.g. DynamoDB [54]) [51]. If the system does not receive any new updates, Eventual Consistency assures that all replicas will eventually converge to the same state. This is why queries can return stale data, or messages can get lost due to a system failure or network partition. Eventual Consistency is therefore more of a liveness guarantee instead of a safety one [53] [51, p. 12].

2.3.2 Chain Replication

Chain Replication (CR) is a simple replication protocol that provides Strong Consistency, more specific Linearizability [48] while offering thread scalability and maintaining high throughput and low latency [49]. It is a special case of the primary/backup replication protocol (also known as passive replication) [55] in which the primary node sends *write-requests* to all backup nodes, waits for their acknowledgments, and then returns to the client while doing local reads for *read-requests*. In the CR protocol, all nodes are connected in a chain of length c , in which the first node is the head node, and the last node is the tail node. Compared to the primary/backup replication, not every node has a connection with the leader (primary) node. Between head and tail, there can be several middle nodes. Therefore, the replication factor depends on c . For a new entry to be replicated, it must traverse the whole chain from head to tail. CR offers a better load-balancing and throughput than the naive primary/backup replication since read/write requests go to the head or tail node, respectively. Therefore the leader's node responsibilities are split between these two nodes. One limitation of the CR is that only the tail node may answer *read-requests* [49].

2.3.3 Chain Replication with Apportioned Queries

Chain Replication with Apportioned Queries (CRAQ) [56] tries to solve those previously mentioned limitations of CR by enabling every node to do local reads. This is done by adding a state to each entry which represents if the entry has already been committed. In the case of CR, that is when the entry has been replicated on the tail node. The tail node sends a message backward through the chain, so all nodes know that the entry is committed and set the state, respectively. This enables the node to answer read requests for this entry by reading locally. As Gavrielatos *et al.* [48] show, local reads do improve the overall performance through load-balancing *read-requests* over all nodes in the chain. In read-heavy workloads, most of the entries will be clean and served by all nodes locally. Therefore, the throughput will increase linearly with the size of c . In write-heavy workloads, the performance remains at least the same [56].

3 Design

In this chapter, we introduce *Ikaria*, an append-only shared log designed for read-heavy workloads and intra-datacenter deployments which incorporates the CRAQ replication protocol [56]. The system consists of multiple servers (referred to as nodes), where the log is replicated. The actual log is stored on Persistent Memory on every server to leverage the low latency and high bandwidth. Further, *Ikaria* makes use of user-space networking by using the eRPC network library [47].

The following sections present the API of *Ikaria*, give an overview of the overall design, describe the different system components in detail, explain the provided consistency models and the failure recovery.

3.1 *Ikaria* API

A client can access *Ikaria* with the for logs essential operations:

- *append*(data) → logPosition : The *append-request* adds new data to the end of the log and returns the associated logPosition for later accessing the log entry again.
- *read*(logPosition) → data : The *read-request* returns for a given logPosition the associated data.

Clients can interact with the log by sending *read-requests* to every node in the chain and *append-requests* to the head node. The *append* operation only returns after the data has been committed, which means the data has been replicated on all nodes and therefore provides Strong Consistency. *Ikaria* offers two different consistency levels for the *read-request*: Strong Consistency and Eventual Consistency. The provided consistency models are described in more detail in subsection 3.2.2.

3.2 System Design

Our System is based on the layered architecture style [57]. *Ikaria* is divided into the following three layers:

- Log Layer
- Replication Layer
- Network Layer

We designed our log with a layered structure since flexibility and adaptability have been a design goal. It makes it easy to use another replication algorithm or another networking library on top of the application. In Figure 3.1 a system overview of *Ikaria* is given. As a storage medium, we use Persistent Memory. We designed our log to use continuous persistent address space to store and retrieve log entries. The Log Layer offers an abstraction for persisting and retrieving such log entries. The Replication Layer manages the log consistency and provides interfaces for the Network Layer and the Log Layer. As we use CRAQ as a replication algorithm that is based on CR, all nodes need to be connected in a chain. Therefore, the Network Layer establishes the connection with the other nodes: the predecessor node, the successor node, and the tail node. It is also responsible for receiving and sending messages via the network library eRPC [47].

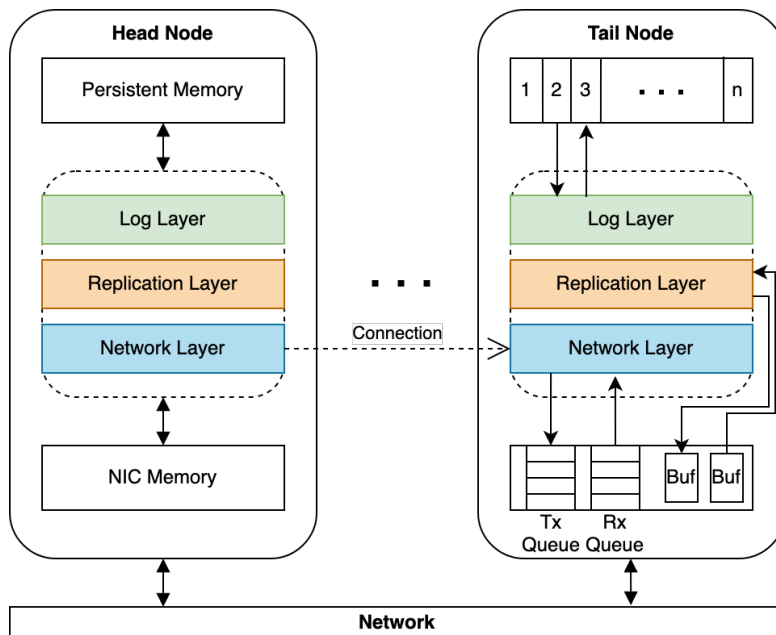


Figure 3.1: *Ikaria* System Overview

A more detailed figure of the software architecture of *Ikaria* is shown in Figure 3.2. As explained before, we incorporate a layered architecture for our software design. The Network, Replication, and Log Layer and their components are grouped in the *LogStack* component as shown in Figure 3.2. The *LogStackManager* manages the *LogStack* by initializing and terminating it. It is also responsible for passing on requests of the clients to the *LogStack*. Classes connected with a dashed arrow in Figure 3.2 imply a dependency between them. For example, the *LogStackManager* class depends on the *Replication* class for passing on requests. The design and functionality of the individual components are described in the following sections in detail.

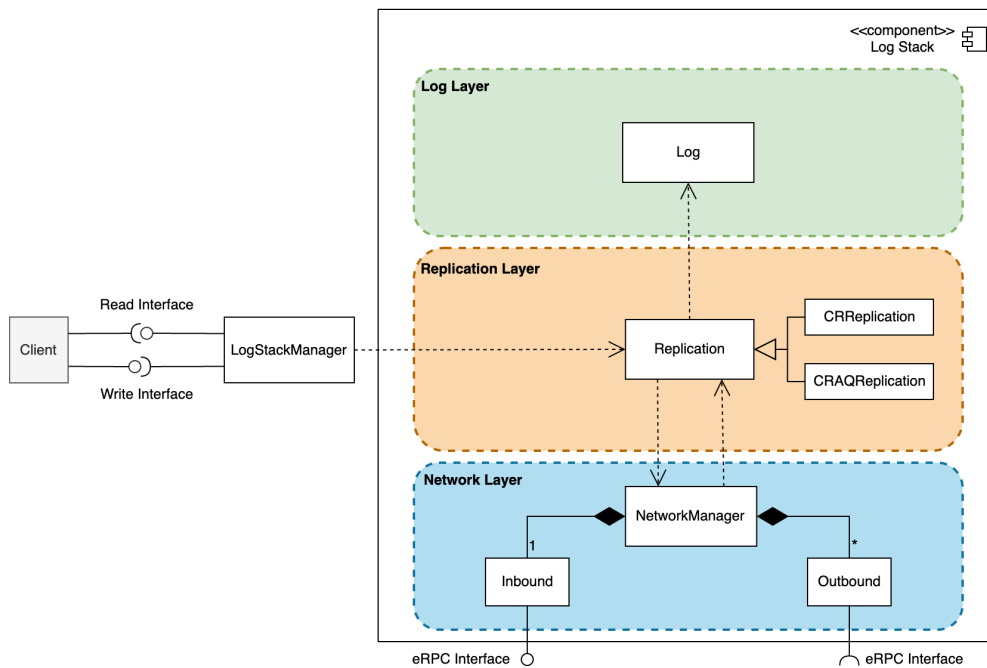


Figure 3.2: *Ikaria* Software Architecture (UML Component Diagram)

3.2.1 Network Layer

The Network Layer is responsible for establishing connections with other nodes and for handling incoming and outgoing messages. For communication between the chain nodes, we make use of the current advancements in network research. We design *Ikaria* to function with asynchronous user-space networking. Therefore, we make use of eRPC

[47] as an asynchronous network library. eRPC handles packet loss and, therefore, provides reliable packet I/O which *Ikaria* relies on [41].

As shown in Figure 3.2 the Network Layer consists of three classes: the *Inbound*, the *Outbound*, and the *NetworkManager* class. The *Inbound* class handles incoming messages via the request handler and sends the associated responses. The *Outbound* class establishes a connection to another node and sends messages respectively receives the responses via the continuation callback function. One instance of this class corresponds to one open connection to another node. So for every needed connection, another instance of the *Outbound* class has to be created. Since *Ikaria* is based on CRAQ, a connection to the predecessor node, successor node, and the *Tail* node is needed. The *NetworkManager* manages these connections and distributes incoming and outgoing messages which it receives from the previously outlined classes or the Replication Layer. The *NetworkManager* is, therefore, the interface between the Network and Replication Layer.

All data which is sent between the nodes has a header attached to it. The in-flight header is shown in Figure 3.3 and includes besides the `messageType`, the `logPosition` of the `logEntry` struct (see subsection 3.2.3) which is sent as payload as well. The `messageType` variable is used to differentiate between the different operations whereas the `logPosition` variable is needed by the Replication Layer. In the following section, we go over the Replication Layer and explain the workflow of the different operations.

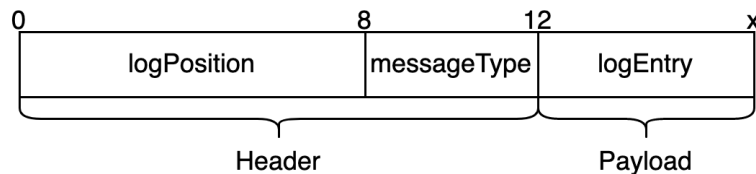


Figure 3.3: In-flight data packet struct

3.2.2 Replication Layer

The Replication Layer incorporates the logic of the replication algorithm. As our goal is to design *Ikaria* for read-heavy workloads, we use the CRAQ algorithm (described in subsection 2.3.3). As CRAQ enables nodes to read locally, superior read load balancing over the original CR protocol is archived [56]. Since local reads might violate the consistency model each `logEntry` has a state attached to it (see `logEntry` header in

Figure 3.9), which is either `error`, `dirty` or `clean`. The `dirty` state implies that an entry is written but possibly not committed yet. The state `clean` marks an entry as committed. All other `logEntries` are marked with the `error` state which is the default state. This means that the entry has not been written yet. We outline our rationale behind this design choice in subsection 3.2.3.

An entry is considered as committed as soon as it is replicated on the *Tail* node. Additionally, we extend the CRAQ protocol by an uncommitted read-request. Compared to the read-request offered by CRAQ, the uncommitted read-request offers only Eventual Consistency. The different request types and their workflows are described in the following sections.

The Replication Layer (see Figure 3.2) consists of the *Replication* class. As shown in Figure 3.2, the Replication Layer is instantiated by the *LogStackManager* (visualized through the dashed arrow). The Replication Layer instantiates the Network and Log Layer classes as it is the core of the system. The *Replication* class offers two functions for each `messageType`. One function for dealing with incoming requests (e.g. `append`) and one for processing received responses for previously sent out requests (e.g. `append_response`). We need two separate functions since we use asynchronous networking and, therefore, a response is not directly received after the request is sent out. The logic in these functions differs depending on the node type: *Head*, *Middle*, and *Tail*. In the following sections, we describe for every message type how the message is sent through the chain and the flow of the message through the node layers. After this, we discuss the provided consistency properties.

Append-Request

When a client wants to append to the log, it sends a message to the *Head* node, where the appends get serialized by assigning a sequential and dedicated `logPosition`. Once an append is serialized, there are only two outcomes: It will be committed, or it will be lost due to a node failure, and the `logPosition` will be filled with a junk value (see section 3.3). When a node receives an *append-request*, it writes the data to its local log at the given `logPosition` and marks it as `dirty`. After this, it forwards the request to its successor. Since every *append-request* gets a unique `logPosition` assigned at the *Head* node, multiple threads can write on a node's log concurrently without interfering with each other. When the *append-request* reaches the *Tail*, it replicates the entry as well and sets the state of the entry to `clean`. After this, the *Tail* sends an acknowledgment (*append-response* in the following) reversely down the chain to its predecessor node. When a node receives such an acknowledgment message, it marks the associated entry as `clean`. The *Head* acknowledges the *append-request* since it already holds an

established connection with the client. The *Head* replies directly to the client, even if not all previous `logPositions` have been acknowledged yet. Figure 3.4 displays the message flow of an *append-request* (black arrows) for some data `A` at `logPosition x` as it propagates through the chain. In every figure, the *Middle* node is exemplary and could consist of multiple *Middle* nodes.

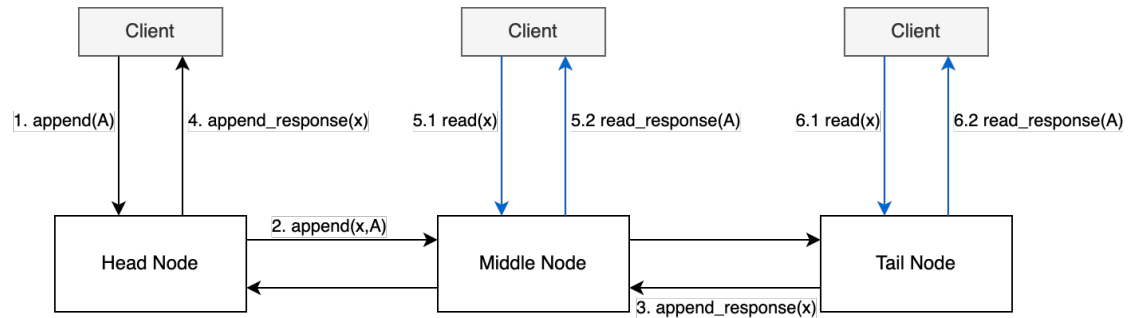
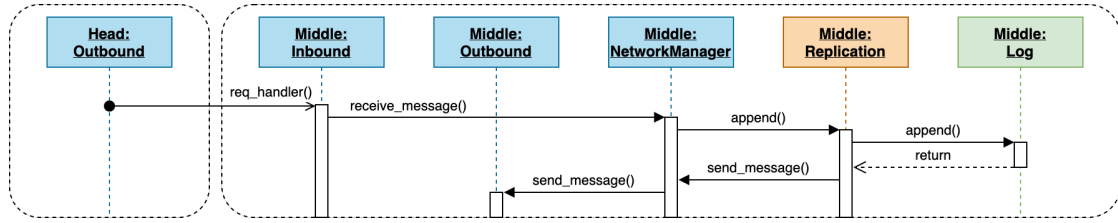


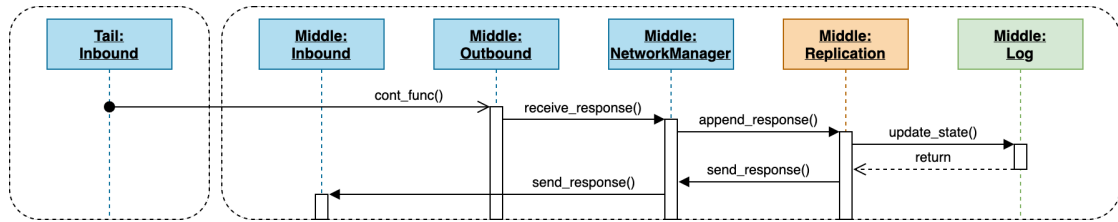
Figure 3.4: *Append-request* Message Flow through the chain (UML Collaboration Diagram)

Even if the *Head* or another node crashes, no already acknowledged message is lost as described in section 3.3. How we deal with lost *append-request* due to node failures is explained in section 3.3 as well. Since we are assuming that all messages arrive at each node eventually (see subsection 3.2.1) the log will only contain holes temporarily. Due to the serialization of *append-requests* at the beginning, the *Head* replying directly to the client after receiving an acknowledgment and the fact that we present an append-only log, we eliminate a requirement the original CR protocol had: a FIFO link between the nodes. This way, we can send messages through the chain concurrently regardless of the `logPosition`, which enables performant multi-threading.

When a node receives an *append-request*, a sequence of functions is called. The sequence diagram in Figure 3.5 visualizes the message flow through the different layers and classes. After the *Inbound* class receives the message via the request handler, the *NetworkManager* calls the dedicated `append` function in the *Replication* class. The *Replication* class delegates the message to the Log Layer, which writes the entry to the log. After the message is persisted, the node forwards the request to its successor node, which is done by the *Outbound* class. If the node is the *Tail* node, it does not forward it to its successor but prepares the *append-response* message which includes the assigned `logPosition`. It then sends the *append-response* via the *Inbound* class to its predecessor.

Figure 3.5: *Append-request* Message Flow through a node (UML Sequence Diagram)

The sequence diagram of an *append-response* is shown in Figure 3.6. First, the continuation function is triggered by the eRPC library in the *Outbound* class. Then the *Replication* class updates the state of the entry from dirty to clean by calling the offered state update function of the Log Layer. Then the message is sent by the *Inbound* class to the predecessor node by responding to the corresponding previous received *append-request*.

Figure 3.6: *Append-response* Message Flow through a node (UML Sequence Diagram)

In *Ikaria* Strong Consistency guarantees that all *append-requests* are sequentially ordered and not acknowledged until they are committed. The ordering is fulfilled since all *append-requests* have to be sent to the *Head* node where they get a unique `logPosition`. The second guarantee is fulfilled since the *Head* sends the acknowledgment to the client only after the *Tail* node has replicated the entry. Therefore, the *append-request* guarantees Strong Consistency. If the *Head* sends an acknowledgment for a committed *append-request* for `logPosition` o to the client, it is not guaranteed that every `logPosition` smaller than o has been acknowledged already. This is not a problem for keeping the Strong Consistency guarantee since acknowledged entries will not be deleted from the chain due to e.g. a node failure. The same assumption is made by other shared logs (e.g. CORFU [6]) as well.

Read-Request

When a client wants to read an entry from the log, it can send a *read-request* to every node in the chain. This is possible because every node knows the state of a specific `logEntry`. When the requested entry is marked as `error` or `clean`, the node can answer the request directly as shown in Figure 3.4. In case the entry is marked as `dirty`, the node sends a small *state-request* to the *Tail* to check if the entry has been committed by now. This happens if either the *append-request* has not reached the *Tail* yet and/or the acknowledgment message for this entry is still somewhere in the chain and has not reached the node yet. The *state-request* only includes the `logPosition` of the requested entry, therefore, it is relatively small. *Ikaria* does not wait for the response by the *Tail* but continues to process further requests. When the *Tail* receives the *state-request* for a `logPosition`, it checks if the corresponding `logEntry` has already been committed. If this is the case the *Tail* returns `clean` to the requesting node. In the case of a log entry's specific *append-request* has not reached the *Tail* yet, the *Tail* answers with the state `error`. An entry on the *Tail* node is never `dirty` as every entry is committed as soon as the entry is persisted on the *Tail*. As explained in the following subsection 3.2.3, we assume that the state of all not written entries is set to `error` which is how the *Tail* can find out if the entry has already been written. Depending on the returned state, the node either returns with its local read value since the requested entry has been committed by now, or it answers with an error message for retaining the Strong Consistency guarantee since the entry has not been fully replicated yet. This procedure is visualized in Figure 3.7.

After discussing the message flow of the *read-request* through the chain, we present the message flow through a node in the following paragraph. The sequence diagram of a *read-request* is modeled in Figure 3.8. When such a request arrives, independently of the node type, the *Replication* class asks the Log Layer for the requested entry. If the state is `error` or `clean`, the node can directly answer the client since the message has not arrived yet or is already committed. In the case of the state being `dirty`, the *Replication* class creates a new *state-request*, for asking the *Tail* about the current state of the entry. The Replication Layer forwards this request to the Network Layer, which transmits the message. On the *Tail*, the *state-request* function asks the Log Layer for the specific entry and returns as a *state-response* the current state to the requesting node. This is not visualized in Figure 3.8 directly but the sequence of invoked functions is similar to reading a `clean` entry. When the *Replication* class of the requesting node receives the *state-response* it acts accordingly. If the state is `clean` the *Replication* class calls the Log Layer to update the entry's state and answers the previous receive *read-request* with its local entry. Otherwise, the Replication Layer returns an error message, which means

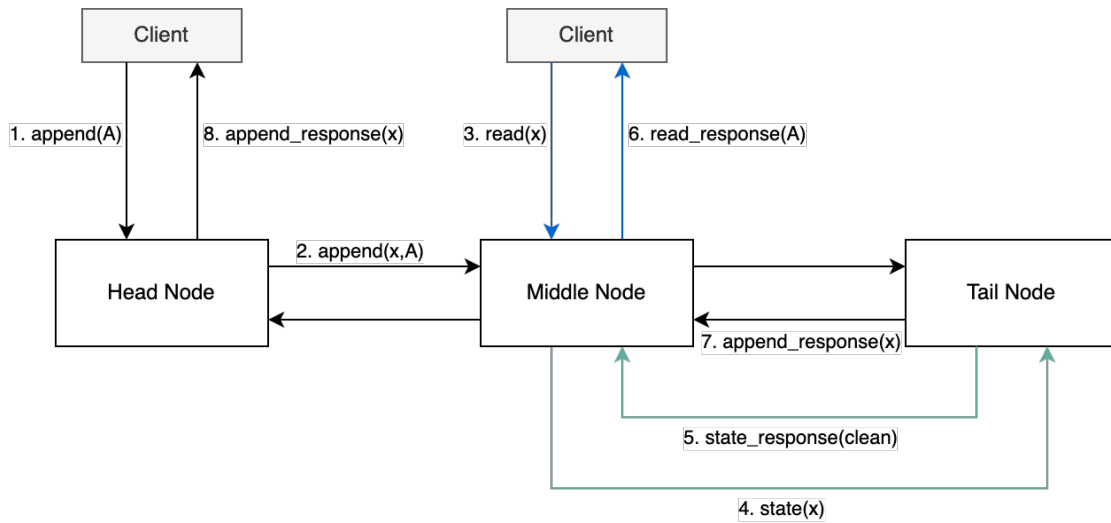


Figure 3.7: *Read-request* Message Flow through the chain (UML Collaboration Diagram)

that the entry does not exist yet or is not replicated on all nodes.

Strong Consistency for the *read-request* guarantees that every node is returning the latest state known by the *Tail*. Since all *append-requests* are only committed when they have reached the *Tail* it will never be the case that two *read-requests* to different nodes in the chain return two different answers. This is ensured through the *state-request*, where each node can request the current status of a specific `logEntry`, as mentioned before. By asking the *Tail* node first if the entry has already been committed, it can be ensured that the most up-to-date value is returned. Therefore, the *read-request* guarantees Strong Consistency.

Uncommitted Read-Request

Besides the normal *read-request*, *Ikaria* offers an uncommitted *read-request* as well. This *read-request* type only holds Eventual Consistency as some systems do not rely on Strong Consistency [54]. When a node receives such a request, it checks locally if a `logEntry` has been written for the requested `logPosition`, independently if the state is dirty or clean. If this is the case, it directly returns the entry to the client. Otherwise, it returns an error message. Compared to the Strong Consistency providing *read-request*, the overall performance should improve by eliminating the need for *state-requests* to the *Tail*. Therefore the latency for this request should be lower as well.

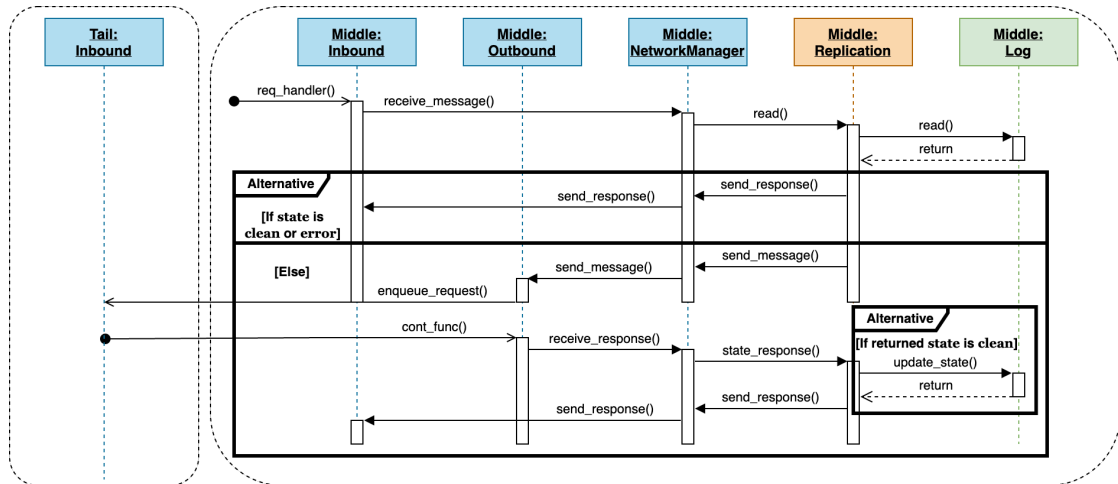


Figure 3.8: *Read-request* Message Flow through a node (UML Sequence Diagram)

The message flow on a node is the same as a local read for a `clean logEntry` which is shown in Figure 3.8. The message flow for the uncommitted *read-request* does not differ between the three node types: It asks the Log Layer for the requested `logPosition` and returns whatever the Log Layer returns to the client.

Eventual Consistency guarantees that a node which receives an uncommitted *read-request* returns the `logEntry` regardless if its current state is `dirty` or `clean`. When the entry's state is `dirty`, the *state-request* is omitted, and the latency of the request is lower. During a chain reconfiguration phase in the case of a *Tail* node failure (see section 3.3) *read-request* can still be answered for entries with state `dirty`. Two *read-requests* sent to two different nodes in the chain could return different results in the case the initial *append-request* has not reached both nodes yet. Therefore, Monotonic-Read Consistency is only satisfied if a client sends the *read-requests* to the same node [56]. It could also happen that a client reads an entry from the *Head* node before the *Head* has forwarded the request to its successor. In the case of a *Head* failure this entry would be lost, which is in line with the definition of Eventual Consistency [48, 53].

3.2.3 Log Layer

The main task of the Log Layer is to persist data, and it offers functionality for reading, writing, and updating entries to the Replication Layer. We introduce the dedicated *Log*

class to achieve a clear separation between the storage library and our implementation so that it can be easily exchanged. The main task of the *Log* class is therefore to provide an abstraction of the storage library. Since we need to make sure that a `logEntry` is persisted before forwarding the request, we need a persistent storage medium. Before PM existed, flash drives have been used for this purpose [6]. Since we have to persist the entries, any storage type which provides better access performance and latency should provide better performance and latency for the shared log. That is why we design our log for Persisting Memory as a storage medium. As storage library we are making use of PMDK [29] (see subsection 2.1.1) for managing the PM. Due to our layered approach, the Log Layer can be exchanged easily for another implementation if needed e.g. an implementation based on the Storage Performance Development Kit (SPDK) [58].

The log itself resides on every node in the chain, and we assume that it lies continuously in the address space. Each entry is persisted in the `logEntry` struct which is shown in Figure 3.9. It includes the `popcount` value, the state of the entry, the length of the data, and the actual data. Each `logEntry` has a fixed size of l bytes, which has to be determined at the application start. This makes it possible to access a specific `logEntry` in $\mathcal{O}(1)$ since the memory address of an entry can be calculated by multiplying the `logPosition` with l .

We require the log file to be zero-filled before being used. This assumption is reasonable since it is a common practice in database systems [59]. This enables every node to check if an entry has already been written by just reading locally. If the value of the state of the entry is zero, which is equal to the error state, the node knows that this entry has not been written yet. Another reason for this requirement is that we use the approach presented by Van Renen *et al.* [59] as `logEntry` design. By adding the bit count of the header and the payload to the `logEntry`, the presented approach is a way to persist an entry to PM by just flushing once. Normal procedures use a second flush to add e.g. a checksum at the entry's end to signal that the entry is consistent. Van Renen *et al.* [59] show in their work that flushing an additional time to PM brings significant performance penalties, which we want to avoid.

When appending an entry, the population count `popcount` value (number of set bits) of the `dataLength` header entry and the data itself is calculated. The state is not part of `popcount` calculation. The reasoning behind this is that we do not have to update the `popcount` value when we update the state e.g. setting it from `dirty` to `clean`. Since the state is updated atomically, and the state is checked by the *Replication* class it is not necessary to include the state in the calculation. It is important to note that only the actual size of the entry is written to the log and not the total fixed entry size. This

ensures that even with the given fixed size *Ikaria* does not lose any performance when variable size entries are persisted.

Since the log file is zeroed and the flush of the popcount value is atomic, a local read can always verify if an entry has been completely written or not [59]. This is necessary since a write of an entry to the log is not atomic and could be interrupted by read access from a different thread. With the addition of the popcount value and the zeroing of the log file before the system starts processing requests, we can assure the consistency of the entries at all times. The *Log* class checks through the popcount value if the data from an *logEntry* is consistent. The popcount value is correct when the entry is not written yet since all bytes are zero as is the popcount value. In this case, the *Replication* class can find out that the entry has not been written yet. Since the error state is equal to the value zero, the *Replication* class can verify that the entry is empty and does not need to ask the *Tail* for the state of the entry. This way, concurrent writes, due to the distinct *logPosition*, and reads are possible and safe, and therefore the log can be accessed concurrently.

Calculating the bit count value might be less efficient than flushing twice if the entry sizes are getting bigger. Van Renen *et al.* [59] showed that it is faster up to at least an entry size of 512 B, especially when the size is aligned to a multiple of the cache line size. As we argued before, PM is well suited for smaller entry sizes than the standard Linux page size of 4096 B because of the byte addressability. That is why we decided to use the single flush approach.

Since the current PM hardware does not perform well when accessed by multiple threads concurrently [25], multiple PM modules can be used. To not exhaust one PM module, the log can be split over n PM modules with a simple round-robin scheduling. Each module gets an identifier starting from 0, counting up which orders them in total order. The responsible module for a specific *logPosition* o can be calculated by: $o \pmod n$. The memory address for the *logEntry* is therefore: $\lfloor o/n \rfloor \cdot l$.

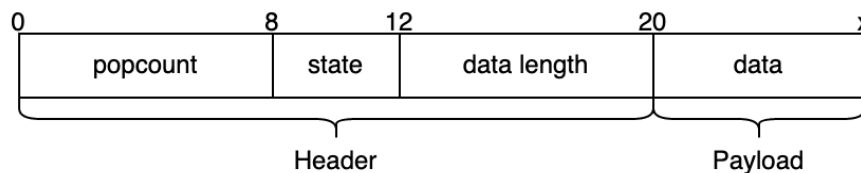


Figure 3.9: *logEntry* struct

3.3 Failure Recovery

Providing local reads comes at a price. The premise for offering local reads is that it has to be ensured that every *append-request* reaches every node and not just a quorum of nodes [48]. If one node in the chain or a network link between two nodes fails, the *append-requests* can not be processed until the chain has been reconfigured. *Read-requests* for already replicated log entries can still be served by the other nodes in the chain. Since a node which discovered that its predecessor/successor failed, does not know the predecessor/successor of the failed node, some kind of coordination service [11, 60] for the node membership management is needed [49, 56]. The coordination service keeps a connection with all chain nodes and reconfigures the chain when a node fails. In the following, the chain recovery will be described and explained how the Strong Consistency guarantee can be assured.

In *Ikaria* there are three types of possible failure scenarios:

- Head failure
- Middle failure
- Tail failure

Head failure: When the *Head* fails, it will be removed from the chain, and its successor will become the new *Head* node. The new *Head* checks locally for holes in the log and which entry has the highest `logPosition` regardless of the state. It could be the case that the entry at `logPosition o` has been lost due to a network failure between the old *Head* and the new one or has not been sent at all due to a node failure, but the entry at position $o + 1$ is in the local log of the new *Head*. Since the old *Head* failed it cannot resend the entry o . This means that this message is lost. Because the new *Head* has not received o , no other node has received it either, especially not the *Tail*. It, therefore, creates an *append-request* for all these missing log entries and fills them with junk values. The new *Head* then sends these requests to its successor as usual. This way, the log will not contain any holes due to lost messages. The new *Head* needs to find the highest written `logPosition o` since it has to set its serialization counter to $o + 1$ for assigning new incoming *append-requests* unique `logPositions`. Now the new *Head* can start processing *append-requests* as normal. The client that has sent one of the *append-requests* which have been lost cannot tell the difference if its message is ignored or maybe just lost on the way to the *Head* node, so it will just retransmit it. This complies with the Strong Consistency guarantee since no log entry is deleted, which has already been committed (or could have been). Therefore, no *read-request* could have been already answered for such an entry.

Middle failure: If a *Middle* node fails, it will be removed from the chain. The predecessor and successor of the crashed node need to establish a connection to have a connected chain again. This is handled by the previously mentioned coordination service. As soon as the two nodes have an active connection, the predecessor has to resent every log entry with the state *dirty* to its new successor. These are the messages which are not fully replicated yet. Therefore, it could be the case that one of these messages has not been forwarded by the failed node. Resending all these messages can assure that no message is lost and they will be committed. As an optimization, the successor could resend the last few acknowledgments for committed entries to the predecessor. Since these messages could have been lost due to the node failure as well, it would enable all nodes up to the *Head* to answer *read-requests* locally. This is not necessarily needed because if the node receives a *read-request* for an entry that is still marked *dirty*, it can always send the *Tail* node a *state-request* for finding out the current state.

Tail failure: When the *Tail* fails, it will be removed from the chain, and its predecessor becomes the new *Tail* node. The new *Tail* sets the state for every log entry which is replicated in its local log and has currently the state *dirty* to the state *clean*. Since it is the new *Tail*, the messages are automatically committed when they are replicated in its local log. For all the new committed entries, it sends the acknowledgment messages to its predecessor as usual. This has to be done since the *Tail* could have committed new entries $o_{committed} + x = o_{new}$, answered *read-requests* for o_{new} , but has not sent the acknowledgment message for o_{new} , which, therefore, has not reached the new *Tail* node (the previous predecessor of the old *Tail*). No messages are lost, due to the implicit requirement by CR, that every message which has reached the *Tail* must have reached its predecessor before. This satisfies the Strong Consistency guarantee.

4 Implementation

After we presented our design for *Ikaria* in the last chapter, we now describe our approach for the implementation. The application is written in C++ and is based on the C++17 standard. We make use of three external libraries namely eRPC [47], PMDK [29] and libpopcnt [61]. eRPC is the used network library, PMDK offers APIs for interacting with persistent memory (see subsection 2.2.3 and subsection 2.1.1), and libpopcnt is used for calculating the popcount value.

One general requirement for the implementation is high performance. Therefore we reduce the need for copying and moving objects/data, reduce the number of expensive system calls, and the use of virtual functions. Due to the lack of virtual functions, there are no C++ interfaces in use. Therefore the layers do not provide actual services, making them easier to replace, but an attempt was made to keep the interfacing functions as similar and clean as possible. The client-side of the application is not a part of this implementation. For the evaluation of our implementation, the messages are generated in the Replication Layer itself, which is described in chapter 5. In the following sections, we describe implementation-specific details and how we incorporated the associated libraries.

4.1 Network Layer

As mentioned in the design chapter, we use eRPC [47] for transferring the messages between the nodes. The eRPC library provides an abstraction for the lowest network layers and offers a simple API for managing messages asynchronously. For initializing eRPC and creating an *RPC* object, a *Nexus* object has to be created which is done in the *LogStackManager* class (see Figure 3.2). The *LogStackManager* class is the first class to be initialized, and this way, it is easier to distribute a reference of the *Nexus* object to the other classes. The *RPC* object is then created in the *NetworkManager* class. As mentioned in section 2.2, every application thread has to create an *RPC* object with a unique *eRPC-id*. This object can then be used to establish connections and *enqueue/receive* requests. A user can create various request types by registering a request handler for every message type as the next step. Although we need more than one message type, we choose to only register one request handler with a generic message type in the *Inbound* class. We choose this approach because the code for different incoming

message types in the *Inbound* class is identical since it only creates the message struct and passes it to the *NetworkManager* class. We avoid redundant code by not registering a request handler for every message type but using only one for all message types. After registering the request handler, the sessions with the other nodes are established by creating multiple instances of the *Outbound* class which sets up the connections.

When a new message is received, the request handler of the *Inbound* class is called by eRPC [47]. For every new arriving request, a message struct is created. This struct includes all vital information like the *messageType*, a handle for the received request for sending a response later on, and the request and response *MsgBuffer* with their current fill sizes. The information in the message struct identifies a request. It is passed between the different classes and layers and exists for the request's lifetime, which is until the response is sent. A response for a previously received message is sent via the *Inbound* class.

When sending a message to another node via an *Outbound* class, we can pass a context variable to eRPC [47]. This way, we can identify the message again when the callback function is invoked when a response is received. As a context variable, the previously mentioned message struct is used. As soon as a response for a previously sent message is received, the continuation callback function is triggered by eRPC [47] which is part of the *Outbound* class as well. In the callback function, the request can be identified through the context variable, and the *NetworkManager* can act accordingly.

Since the eRPC library [47] is an asynchronous framework, we need to run an event loop for sending and receiving messages. The event loop is called whenever a response is received, or a message is sent to another node. This ensures that the messages are sent out immediately, and the latency is kept as low as possible.

4.2 Replication Layer

We have implemented two different replication algorithms: Chain Replication and CRAQ. These classes can be used interchangeably through the use of C++ templates [62]. The uncommitted *read-request* is currently implemented in the CRAQ class. We implemented the Chain Replication class for comparing it later in the evaluation with *Ikaria*. *Append-requests* are working equally as the *append-requests* from *Ikaria*, besides the fact that Chain Replication is not dependent on the state. *Read-requests* are only sent to the *Tail* node which answers either with the *logEntry* or with an error message.

4.3 Log Layer

As explained in subsection 3.2.3, the Log Layer of *Ikaria* uses the PMDK library [29] (see subsection 2.1.1) for managing the PM. In the following section, we describe what libraries of PMDK we use and how we extend the *libpmemlog* library for our use case.

4.3.1 PMDK

The PMDK consists of several libraries for different use cases as explained in subsection 2.1.1. For our log, we make use of the *libpmemlog* library, which is written in C. While the offered managing functionality of the PM module is convenient to use, the *libpmemlog* library, unfortunately, does not allow for concurrent writing to multiple `logPositions`. It does not offer a dedicated read method for accessing a specific `logEntry` as well. That is why we extend the *libpmemlog* library by adding three additional functions:

- `pmemlog_write(PMEMlogpool, logEntry, logEntryLength, logPosition)`: Writes a `logEntry` to a specific `logPosition`.
- `pmemlog_read(PMEMlogpool, logPosition)`: Returns a pointer for a requested entry which is associated with a specific `logPosition`.
- `pmemlog_zeroing(PMEMlogpool)`: Writes zeros to the available log space.

The *PMEMlogpool* contains meta information like a pointer to the start offset of the log, information if the storage medium is actual PM and if the PM is used in DAX mode, and so forth. This object is returned when the log is initialized with the `pmemlog_create` function of *libpmemlog* which is called by the *Log* class during its init phase.

The two functions `pmemlog_write` and `pmemlog_read` are performing checks before they are writing/reading an `logEntry`. The `pmemlog_read` function checks if the requested `logPosition` is in bounds of the log and just returns a pointer to the `logEntry`. The `pmemlog_write` function checks the `logPosition` for validity and persists the entry. The function knows through the *PMEMlogpool* object, which is passed as a parameter, if the storage medium is actual PM or not and uses different functions for copying the data. To guarantee that the data is persisted, a cache-line flush must be performed after every write to evict the data out of the L3 cache to the PM module [21]. This is ensured by using the `pmem_memcpy_persist` function [63] from the *libpmem* library. We assume that no *append-requests* overwrites an already written `logEntry` due to the serialization counter assigning separate `logPositions` (see section 4.2). The third function we have added is the `pmemlog_zeroing` function. It writes zeros to the available log address

space before the log is used. As explained in subsection 3.2.3, this is needed for ensuring consistency. How the PMDK library is integrated and how the consistency of the entries is ensured are described in the following section.

4.3.2 Log Abstraction

The *Log* class initializes the PMDK library by creating a new log pool and managing the *PMEMlogpool* object during its construction phase. Therefore the path where the log should be created and the specific log size has to be passed as parameters. After creating the log pool, the `pmemlog_zeroing` function is called for zeroing the entire log address space. When the construction of the *Log* class is done, it is ready for reading or writing entries. The pool is closed with the `pmemlog_close` function in the destructor of the *Log* class.

The `popcount` value is managed in the *Log* class and not in the `pmemlog_write` and `pmemlog_read` function of the PMDK library [29]. One reason is that for calculating the value the function has to know the `logEntry` header Figure 3.9, to write the `popcount` value at the correct place. This would mean that our PMDK extended functions would not be very generic, which is not desirable. If another storage library is used in the future, the `popcount`-feature would also have to be added, which is another reason to place this functionality in the *Log* class.

The C++20 [64] standard introduced a `popcount` function in the standard library. Since we based our implementation on the C++17 standard we use the open-source library *libpopcnt* [61]. Another reason why we choose this library is that it calculates the `popcount` of an array of a given size which is exactly what we need. This library implements some improvements for the calculation of the `popcount` value which are offered by modern processors [65]. After calculating the `popcount` value the extended PMDK function `pmemlog_write` is called for persisting the `logEntry`.

The read operation of the *Log* class retrieves the pointer to the requested entry with the `pmemlog_read` function. It then calculates the `popcount` value for the `logEntry` and matches it with the `popcount` value in the header for checking the consistency. The pointer and the actual length of the entry are returned to the calling function.

The last function offered by the *Log* class is the update state function. It sets a new state with help of the `pmemlog_write` function by writing the new state to an existing entry. Since the state is equal or smaller to the atomic size, which is supported by the processor (in our case 8 B), the update happens atomically.

If our approach for ensuring consistency is not desired, a two-step persisting system could be added as future work. This can be achieved easily through the state in the header. The state has to be set in the first flush to e.g. `error` and has to be changed in a second flush from `error` to `dirty` or `clean`.

4.4 Bootstrap and Teardown

We have added two more requests besides the *append-request*, *read-request* and *update-request*. The first is the *setup-request* which synchronizes the start of operation between the chain nodes. The setup process is started by the *Head* node which sends a *setup-request* to its successor. As soon as a node is finished with its initialization, it either waits for the *setup-request* or, if already received, it forwards the message down the chain. When the *Head* node receives the response, it knows that all nodes are ready and starts processing incoming messages. The other nodes wait until they receive the first request other than the *setup-request* before they start to be operational. If every node would start processing incoming requests as soon as they send the *setup-response* to its predecessor, other nodes would still wait for the *setup-response* and will not process any incoming requests. The first request other than the *setup-request* can therefore be seen as a kickoff for all chain nodes.

The second added request type is the *terminate-request* which works similar as the *setup-request*. When a node receives a *terminate-request* it stops answering requests from clients and only processes messages from other nodes in the chain. As soon as a node receives the *terminate-response* it forwards the response to its predecessor and shuts down. This way, it can be assured that every node receives the *terminate-request*.

The setup and termination process for the chain is the same for every replication algorithm. These two message types have been added to synchronize the startup and termination of the different nodes during a benchmark.

4.5 Multi-Threading

To leverage the existing hardware parallelism, our implementation can be initialized with several worker threads. The threads are managed by the *LogStackManager* class (see Figure 3.2). This class offers functionality for starting a desired amount of threads, terminating them, and retrieving their current status. The shared resources between the threads are the serialization counter in the Replication Layer, the *PMEMlogpool* object in the Log Layer, and the *Nexus* object in the Network Layer. For simplicity, we create

an instance of all layers, the complete *LogStack*, for every thread. Since the *Replication* class initializes the other two layers, the *LogStackManager* class creates several instances of the *Replication* class (see Figure 3.2).

In the constructor of the *Replication* class, a dedicated function is started in a thread that initializes the other layers. We initialize the serialization counter as an atomic integer variable [66] which synchronizes the access between the different threads by design. Every thread accesses the counter with the `fetch_add` function, which increments the atomic variable and returns the value before the incrementation. This way, it is ensured that every new entry gets a unique `logPosition` independently of running the application single or multi-threaded.

Since the *Nexus* object is created in the *LogStackManager* class, it is created exactly once and then passed to the starting threads. The request handler function is registered exactly once at the *Nexus* object in the *Inbound* class. This is ensured by using the `call_once` function [67] which executes exactly one time. Each thread then uses the *Nexus* object for creating the *RPC* object with a distinct *eRPC-id*. This object and the associated *eRPC-id* are unique per thread. It does not matter which thread registers the request handler. It just has to be registered once. Incoming requests specify the *eRPC-id* they are sent to and therefore the *eRPC* library distributes the requests to the thread which created the *RPC* object with the specific *eRPC-id*. Since each *RPC* object has its own queues for transmitting and receiving messages multi-threaded operations are not an issue [41]. As shown in Figure 4.1, the *Outbound* classes from the *LogStack* instance connect to the *LogStacks* on the other nodes which have the same *eRPC-id*. This way, we have the same amount of execution chains as we have threads running on a single node. Every node in the chain must start the same amount of worker threads. Otherwise, some threads can not start processing requests.

The *PMEMlogpool* object is created once by the first *Log* class which is initialized. This is done the same way as registering the request handler by using the `call_once` function [67]. Every application thread therefore writes/reads to/from the same log file. Due to our design of distinct `logPositions` and the `popcount` consistency feature concurrent accesses to the log are possible.

Through this multi-threaded design, we enable *Ikaria* to scale well with an increasing amount of threads, as we show in the following chapter.

4 Implementation

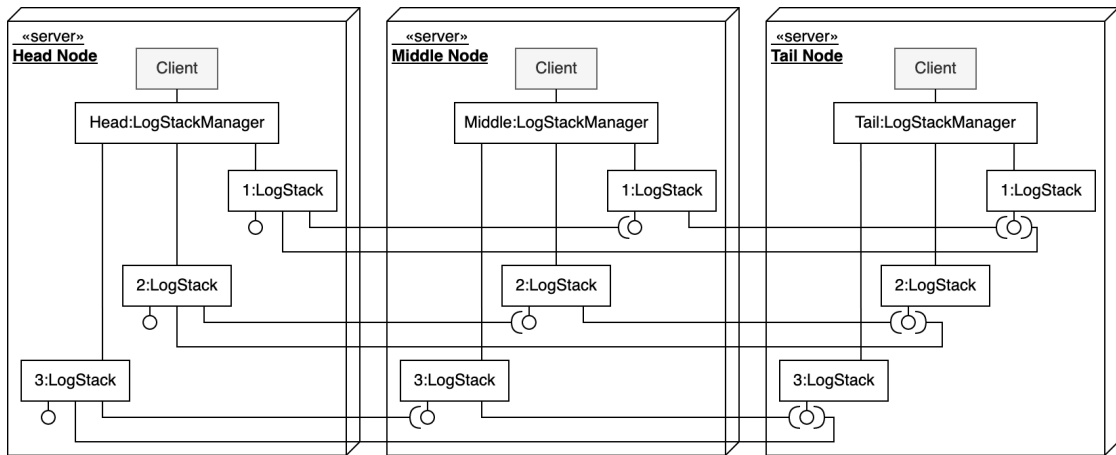


Figure 4.1: Deployment of a three node chain with respectively three instances/threads of the *LogStack* running (UML Deployment Diagram)

5 Evaluation

After presenting the design and the implementation, we now evaluate the throughput and scalability of *Ikarria*.

5.1 Experimental Setup

Our evaluation is performed on a cluster of five servers containing an Intel(R) Core(TM) i9-9900K, with 8 physical cores (16 threads) each and 64 GiB dual-channel memory with 2666 MT/s. Each server is equipped with an Intel Corporation Ethernet Controller XL710 for 40GbE QSFP+ (rev 02). The servers are connected over a 40GbE QSFP+ network switch.

For the evaluation, we are adapting our existing implementation to process requests and generate them. Therefore, every chain or worker thread is seen as a client and has a maximum outstanding request window of 50 messages. This is a similar approach as Terrace *et al.* [56] followed in their evaluation. Another reason we choose a request window is that *append-requests* are a lot slower than *read-requests*. This is trivially true for reading locally compared to replicating a entry on all nodes, but also for reading from the *Tail* since a *read-request* is sent directly to the *Tail*. If we do not limit the messages in-flight, each thread would keep generating messages and pile unfinished *append-requests* since *read-requests* would finish quickly. This way, we would not be able to comply with a targeted benchmark read workload, for example, 90% read messages, since we finish our benchmark after a specific time. Another advantage of this low outstanding request window is that messages are produced more balanced over all participating threads on other nodes. If the request window is larger, the *Head* saturates the other nodes with requests, so they cannot send messages out on their own. Since Chain Replication can only be as fast as its slowest node, we want to split the message generating load over all nodes, so no node becomes a bottleneck. The outstanding request window does not influence forwarding messages from other nodes but just the message generation of the individual threads. As we will show in section 5.2, 50 messages per thread seem enough to saturate the replication chain. Because each worker thread is a client at the same time, every node has an additional connection with the *Head* node since the clients need to send the *append-requests* there.

To generate *append-requests* and *read-requests* not just alternating but randomly while keeping a certain ratio between the two operations (e.g. 90% *read-requests* and 10% *append-requests*), we use a random function for generating equally distributed numbers. By calculating module 100 of the generated number x ($x \pmod{100}$) we can generate message types according to different load patterns, for example, a 90% read-workload. Since the numbers are generated evenly distributed, it is ensured that we benchmark with the targeted read-workload. These random numbers are generated with a Xorshift random generator function [68]. We use this function because the system call *rand* [69] leads to a bottleneck in our multi-threaded benchmark and slows the whole application down. The requested *logPosition* for the individual *read-requests* are selected randomly from an interval between the *logPosition* from the last seen *append-request* and the zeroth *logPosition*. The interval grows continuously with every new *append-request* send. We mostly circumvent caching on the nodes of previous read entries through this interval since the interval quickly outgrows the cache size. The reasoning behind this is to make the results of the different benchmarks comparable. We argue that this could simulate an access pattern of a client who tries to catch up and reads sequentially through older log entries. In the following benchmarks, this access pattern is used besides the benchmark in section 5.4. One problem of reading entries in this huge interval is that the probability decreases drastically for reading a *logPosition* which is not yet acknowledged. Since another valid access pattern by a client could always be to read the newest entries of the log, or in other words, the *Tail* of the log, we evaluate this access pattern in section 5.4.

In the following experiments, we use the maximum number of available threads, which in our case is 16. This is because we use DPDK as a transport layer for eRPC. DPDK enforces only to run one DPDK worker thread per logical core for avoiding costly context switches [70]. This is why the maximum number of threads in the benchmarks is equal to the maximum number of logical cores. If not stated otherwise, each experiment is run for 40 seconds which should marginalize any biases due to the burn-in phase at the beginning. We assume the burn-in phase to be very short due to the small outstanding message window. The benchmarking application can be started at every chain node independently. The different nodes synchronize the beginning of the benchmark with the previously described *setup-request*, so the benchmark start delay is neglectable. After the benchmark finishes, the data from every thread is collected, and the values for all nodes are added together to provide the measurements of the total chain. Since *Ikaria* is designed for read-heavy workloads, we want to evaluate how it performs under read-heavy workloads of 90% and 95% *read-request* share while comparing it to a balanced 50% read-workload. Further, we are interested in evaluating the advantages of local reads for a shared log and how the log scales. Therefore we

compare our implementation with the original chain replication protocol (CR). *Ikarria* with the uncommitted *read-request* is abbreviated as UCRAQ in the following whereas *Ikarria* with the normal *read-request* is abbreviated as CRAQ.

As described in subsection 2.1.2, emulating Persistent Memory is complex and previous attempts seem only to have provided unrepresentative results. Yang *et al.* [25] showed that the emulating techniques used up today are not producing representative results. Unfortunately, we do not have actual Persistent Memory modules at our hands for benchmarking our application. This is why we use DRAM as a storage medium instead and choose the benchmark parameter, which theoretically should leverage the maximum possible access speeds of real Persistent Memory. We leave PM benchmarks as future work. As length for `logEntries`, we choose 256 B since the current PM hardware has the best throughput for this specific size [25]. Since we present an append-only log, the writing of the entries should be sequential. Even if appends arrive out-of-order in the typical case, they have a sufficient write locality because their *logPositions* should not lie far apart from each other. Since the PM module's internal memory controller module contains a write-buffer, as explained in the previous 2.1.2, sufficient locality of the *logPositions* provides sequential write performance [25]. This assumes that the `logEntries`' size does not exceed the write-buffer size so that multiple writes fit into the buffer. According to Yang *et al.* the write buffer has a size of 16 kB, which is another reason why we choose 256 B as `logEntry` size [25]. For the experiments, we compile PMDK with the environment variable `MEM_IS_PMEM_FORCE` set on true, which forces the library to use the dedicated methods for PM instead of other storage types. This way, for persisting entries, the more performant `flush` is used instead of `msync`. Each experiment is run 3 times, and the final result is obtained by averaging the individual experiment results.

To summarize, the following benchmarks, if not stated otherwise, are measured with each node running 16 threads where each has a 50 message outstanding request window and each log entry a size of 256 B. In the following, we will evaluate the chain saturation, the operation throughput, and the scalability of *Ikarria*.

5.2 Chain Saturation

Terrace *et al.* [56] used 50 outstanding messages per client in the original CRAQ paper. Since they used external clients, we want to verify that 50 messages are saturating the 3-node, 4-node and 5-node chain setups. In Figure 5.1 and Figure 5.2 we compare different outstanding request windows from 50 concurrent messages in-flight per thread up to 500 messages over the different chain lengths.

5 Evaluation

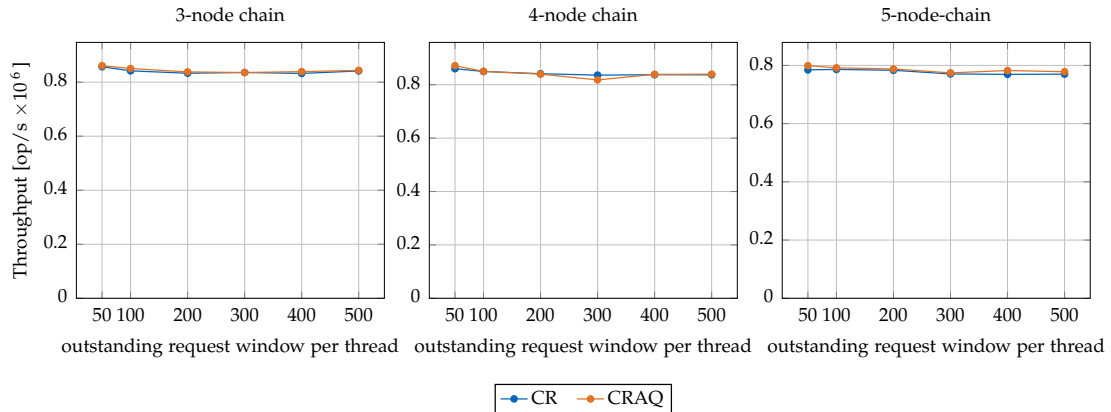


Figure 5.1: Comparison of different outstanding request window sizes for an append-only workload with different plots for the 3-node, 4-node and 5-node chain setup.

We first saturate the chain with an append-only workload (Figure 5.1) and compare CR and CRAQ how the operations per second behave. We would assume that if the chain is not saturated yet, the link between the server and the server itself does not receive enough messages to process and therefore idles. The chain is saturated when one of these two cannot process more messages and is, therefore, the bottleneck. Having more outstanding messages per thread and therefore more messages in the chain, the links between the servers and the servers themselves should process more messages if they are not yet utilized to capacity. As we increase the outstanding request window, we can observe that neither for CR nor CRAQ, the operations per second increase significantly when sending more messages per thread. Therefore we argue that 50 messages are enough for saturating the chain with *append-requests*. We assume that the subtle changes of the values in the plot are due to writing heavily to DRAM. Since memory is used by the other processes on the servers, we assume that this is why the values are not precisely the same. Another interesting observation is that the operations per second vary for the different chain sizes. We explain this behavior in section 5.3. In Figure 5.2 we perform the same benchmark but with an read-only workload. We can observe the same behavior as before: The operations per second do not change significantly during different outstanding request windows per chain setup. Since all other workloads are a mix of *append-requests* and *read-requests*, we argue that 50 messages are enough to saturate every read-workload. We, therefore, follow the approach of the original CRAQ paper by choosing 50 messages per client as our outstanding request window [56].

5 Evaluation

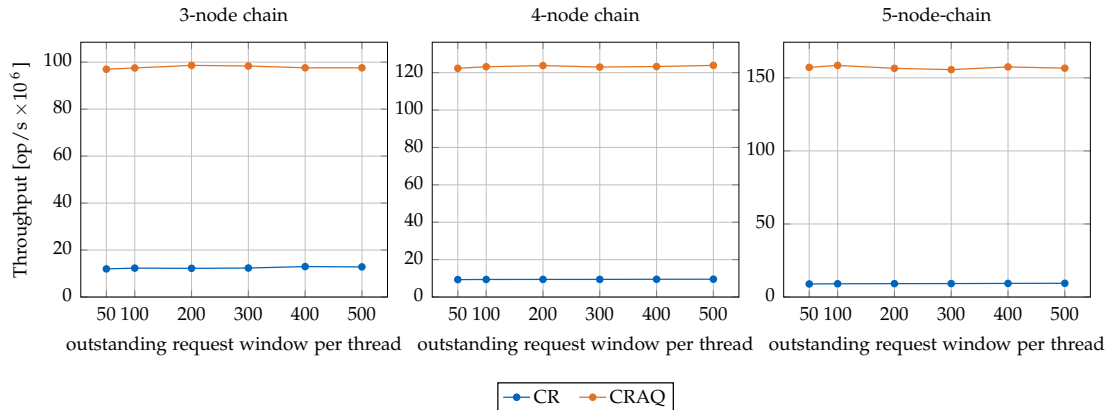


Figure 5.2: Comparison of different outstanding request window sizes for a read-only workload with different plots for the 3-node, 4-node and 5-node chain setup.

5.3 Operation Throughput

In this section, we evaluate the available operation throughput of *Ikaria* by comparing it with the original CR protocol. First, we compare how the replication protocols perform when processing only *append-requests* and only *read-requests* for setting a baseline. These two workloads should be the upper and lower limit of the latter read-workload benchmarks since they consist of mixed message types. In Figure 5.3 the results of an append-only and read-only workload are shown. What we would expect to see for the append-only plot is a subtle decline for the requests when more nodes are added to the chain. This is because a single operation would need to be replicated on more nodes, and therefore, the latency of the operations increases. Increased latency for the operations means less operations per second. We can observe this subtle decline for the 4-node and 5-node setup for all replication protocols as expected. One of the open questions is why this subtle decline is not observable between the 3-node and 4-node setup. Since we showed in the previous section that the chain should be saturated with requests, this cannot explain the observed behavior. These results could be due to our approach to generating messages in each thread on each node, so maybe messages are not well distributed when passing through the chain. Due to the request window, every thread only sends a new message when it receives a response for a previously sent-out message. Since the other nodes need to send their *append-requests* to the *Head* node first (and wait for the acknowledgment from the *Head*), this added delay could help to generate a more evenly load on the *Head* when using a 4-node chain compared to a 3-node chain. We leave it as future work to evaluate the system with external clients

who are not incorporated into the system itself. With this separation of the benchmark from the application, the evaluation could be more targeted, and the actual creation of messages and benchmarking does not influence the system’s overall performance. When comparing the different replication protocols with each other, it seems like that the added access for setting the state of the entry in *Ikaria* does not influence the maximum possible append operations per second (a op/s). We assume that due to the actual writing and persisting of the entries when passing the chain from *Head* to *Tail*, the minor state updates of 8 B (compared to 256 B logEntry size) do not carry weight. This could be due to the usage of DRAM as a storage layer which is quite fast in writing random entries and especially values that do not exceed a cache line size which in our case is 64 B. Since PM is not as fast at writing small values, the log entries’ updates could mean a performance drop. To answer this question, the experiment has to be repeated with actual PM hardware modules, but according to previously made benchmarks, this seems likely [25, 59].

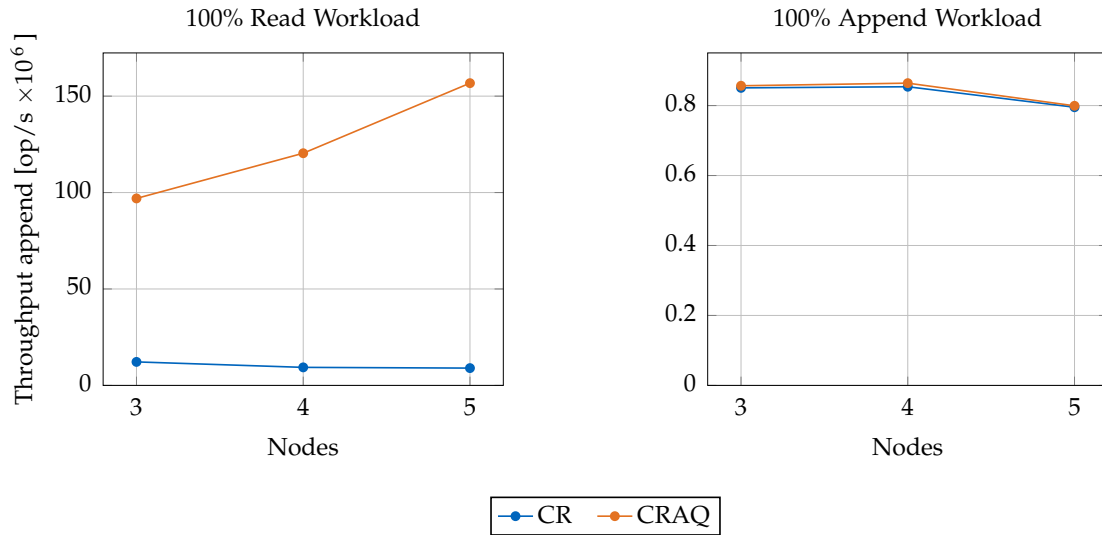


Figure 5.3: Comparison of the different protocols under 100% read-workload and 100% append-workload.

The left plot in Figure 5.3 shows a *read-request* only benchmark and therefore the maximum possible read-op/s (r op/s). Here we can observe a big difference between the CR *read-request* and the *Ikaria read-request*. For a 3-node setup, CRAQ/UCRAQ processes around 100×10^6 r op/s and every node added to the chain increases the total requests for about 30×10^6 r op/s. The values of the CRAQ/UCRAQ are composed of

every node in the chain reading locally, so it shows the added up maximum possible reads a single node can do. We can therefore conclude that a single node can process 30×10^6 r op/s when it does not have to process any other incoming requests. As we can see the results scale linearly with the length of the chain, which is as expected. Therefore *Ikarria* should perform excellently under read-heavy workloads.

In our setup the bottleneck for CR is around 12×10^6 r op/s for a 3-node setup and decreases for a longer chain. Since in CR all *read-requests* have to be sent to the *Tail* node, the natural bottleneck is how fast the *Tail* can read locally and answer the requests. On top of processing requests which are generated on the *Tail* node, it has to process incoming requests from the other nodes as well. Compared to the 30×10^6 r op/s a node can process when using CRAQ/UCRAQ, the *Tail* node can only process less than half the amount of requests. This shows the performance penalty of receiving and transmitting *read-requests* from other nodes. The values are decreasing for an increased size of chain nodes since the *Tail* node processes more external requests than locally generated ones. We can therefore observe a clear bottleneck for the CR, which we will investigate in the following sections.

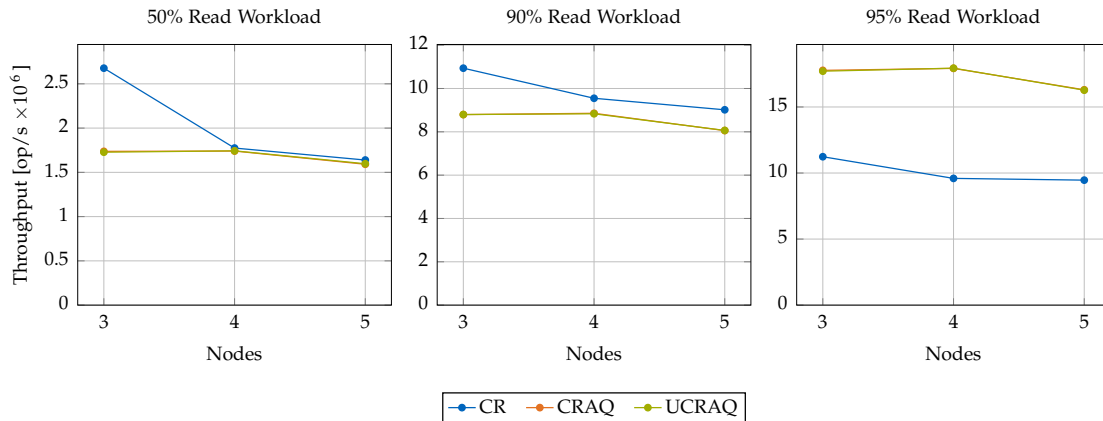


Figure 5.4: Throughput comparison for the 3-node, 4-node and 5-node chain with 50 %, 90 % and 95 % read-workloads.

As the next step, we evaluate the system under different workload patterns with different chain sizes. As explained in section 5.1, we choose 50%, 90%, and 95% read messages as workloads. In Figure 5.4 the throughput results of the different workloads are shown. For CRAQ and UCRAQ, we can see that the curves look similar to the append-only plot from Figure 5.3. The append operations per second are very similar to the append-only plot, which means that the *read-requests* performed locally do not

restrict the processing of the *append-requests*. This is as expected since *append-requests* have a much higher latency than *read-requests* because they need to traverse the chain and be replicated on every node. Since we are binding the amount of processed *append-requests* at the amount of processed *read-requests* by enforcing the read/append distribution, the latency of the *append-requests* is the bottleneck of the benchmark. Even if the node could process a lot more *read-requests*, it is waiting for previously sent-out append messages since its outstanding request window is full. This applies to all three plots in Figure 5.4. We can observe a reasonable throughput for CRAQ/UCRAQ. For the 50%-workload the throughput is around 1.7×10^6 op/s. For 90% and 95%, the throughput scales up to 8.5×10^6 op/s and 17.5×10^6 op/s respectively. These are promising results since we have designed *Ikaria*, especially for read-heavy workloads.

When comparing the CR with CRAQ/UCRAQ in the three plots in Figure 5.4 we can observe that CR outperforms the other two replication protocols, especially for the 3-node setup. While the CRAQ append operations per second stay at the same rates measured in the right plot of Figure 5.3, the append operations per second for CR exceed these values. Since in CR there are *read-requests* in flight as well as *append-requests* compared to CRAQ where only *append-requests* are in flight while *read-requests* are finishing in an instant. As explained before, one reason for this behavior could be the better distribution of the *append-requests*. This would explain why the performance of the chain is only so drastically better when using a 3-chain since the *Tail* does not get overloaded by the *read-requests* from the other chain nodes. Since each node only sends further messages when they receive acknowledgments due to their outstanding request window, it could be that the write load shifts from one node to the next. So some nodes may be idle at certain times while others are saturated. If this is the case, then CRAQ/UCRAQ should perform similarly when benchmarked with outstanding clients, since the clients have to send *read-requests* and *append-requests* to the different nodes when using CRAQ as well. CR can process around 0.5×10^6 a op/s more with 1.3×10^6 a op/s for the 50% read-workload compared to 0.85×10^6 a op/s in Figure 5.3. It is hard to find an explanation for this behavior that aligns with the other benchmarks besides the ones given before. For the 90% and 95% workloads we can observe that CR is at its bottleneck at around 12×10^6 r op/s for the 3-node setup. The *read-requests* become the bottleneck, and CRAQ/UCRAQ starts to outperform CR.

In Figure 5.5 we want to compare the different replication protocols under read-workloads between 50% and 100% for the same chain length. It is essential to mention that the values on the x-axis are logarithmically scaled. Here we can observe how CRAQ and UCRAQ begin to outperform CR at read-workloads which are higher than 90% as we assumed before. The plots show that the bottleneck of CR starts around this read-workload. As seen before, CR outperforms CRAQ/UCRAQ in the 3-node setup for read-workloads smaller than 90%. Compared to the 3-node setup, in the

5 Evaluation

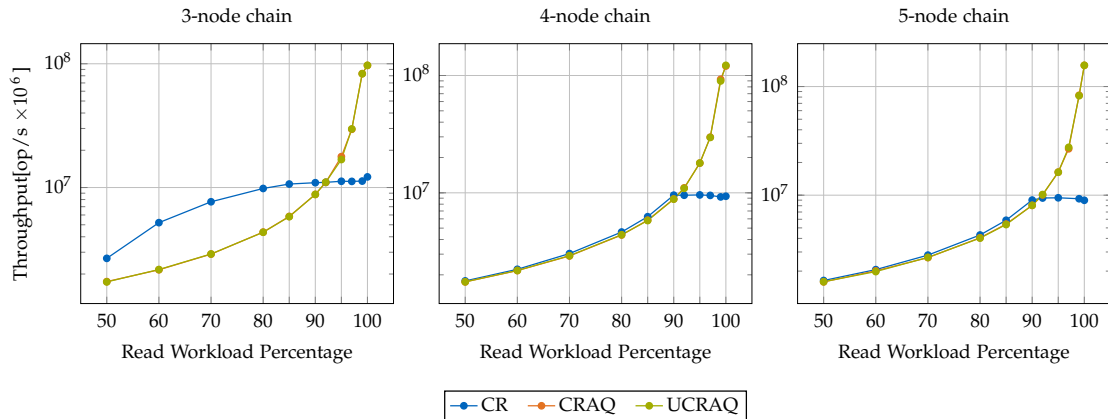


Figure 5.5: Comparison between the replication algorithms for the 3-node, 4-node and 5-node chain and different read-workloads.

4-node and 5-node chain, CR performs similarly as CRAQ/UCRAQ for read-workloads smaller than 90%. The a op/s are growing exponentially for CRAQ/UCRAQ with increasing workloads. Since we logarithmically scaled the x-axis, this means that the a op/s are growing by magnitudes for increasing read-workloads. These are promising results for the CRAQ/UCRAQ replication protocols.

5.4 Tail Reading

As mentioned in section 5.1 we use a random read access pattern for most of our benchmarks. In this section, we want to examine how CRAQ and CR behave when the nodes read a range of the newest log entries. In Figure 5.6 we, therefore, compare the CRAQ/UCRAQ replication protocol with the CR replication protocol in a 3-node chain setup with a 50% read-workload. We denote the size of the read-window as w . When defining the newest seen $logPosition$ in an *append-request* on a node as o , the read-interval can be defined as: $(o - w) \leq x \leq o$, where x is the requested $logPosition$. We only read a range of the w newest written log entries. Figure 5.6 shows the measured throughput over the read-window size w . The size of the interval is shown on the x-axis of Figure 5.6 whereas, on the y-axis, the throughput is shown. We compare the op/s of CR, CRAQ and UCRAQ. Additionally, the plot shows the CRAQ r op/s and the amount of CRAQ *state-requests* sent.

Since the nodes are always sending a *read-request* to the *Tail* node when using CR, the op/s are linear and not changing for different interval diameters. The same applies

to UCRAQ, where a *read-request* is never sent to a different node. We can observe that up to an interval diameter of at least 256 a node has to send a *state-request* for almost every *read-request* when using the CRAQ replication algorithm. The reason why the state-op/s are not equal to the r op/s can be explained because of the *Tail* node not sending any *state-requests* but reading locally. Another interesting observation is that CRAQ suddenly performs a lot better than UCRAQ for an interval size smaller than 4096, and the op/s are closer to the op/s of the CR. It looks like that our assumption made in the previous section is correct. We can conclude that CR performs better than CRAQ due to their additional in-flight *read-requests*. This is also confirmed when looking at reading intervals bigger than 1024. We can see how the performance of CRAQ starts to decrease until the op/s are equally to the op/s of UCRAQ again.

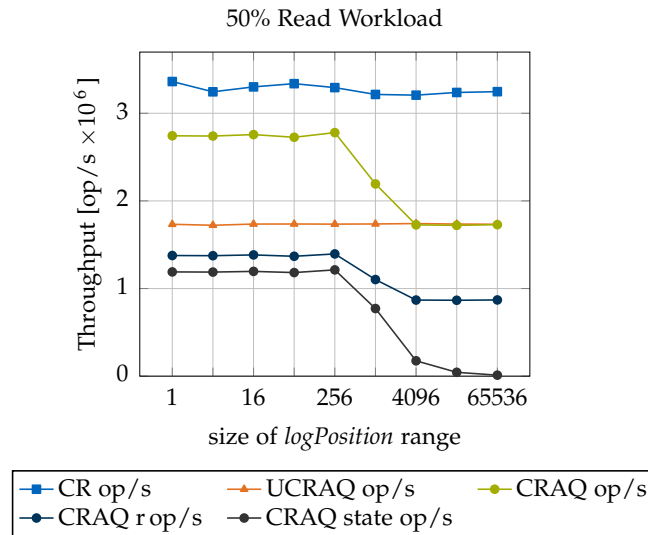


Figure 5.6: Comparison of CR, CRAQ and UCRAQ when reading only recently added *logPositions*. Additionally, the plot shows the amount of r op/s and state-op/s for CRAQ.

To explain why the amount of *state-requests* are staying constant to an interval size of 256, we have to look at the total amount of in-flight messages. We have set the messages in-flight per thread to 50, running 16 threads on a node. So there is a maximum of 800 messages in-flight per node. As described before, we assume that the *append-requests* fill the in-flight message window because they take a lot longer to complete compared to the *read-requests*. From the maximum possible messages per node, a high percentage should be *append-requests*. The total possible unacknowledged *append-requests* in the

chain are therefore the 800 messages in-flight per node multiplied by the number of nodes besides the *Tail* node: 1600 *append-requests*. The decline of the op/s for CRAQ at an interval size of 1024 supports our argumentation. For interval sizes bigger than 1024, the chance for reading a *logPosition* which has already been acknowledged grows, and therefore the amount of *state-requests* declines. We can conclude that when mostly the newer entries are read CRAQ performs similar to CR and we can therefore confirm the findings of Terrace *et al.* [56].

5.5 Scalability

In this section, we focus on the scalability of *Ikaria*. As already mentioned in previous sections, *Ikaria* scales well with more nodes added to the chain. In the following experiments, we use a 4-node chain setup.

Since one goal of *Ikaria* is to build a highly parallelized shared log, we evaluate how well the application scales with an increasing amount of threads per node and therefore show the importance of good thread scalability for replication protocols. In Figure 5.7 we evaluate the protocols while using different amounts of threads during a 50 % and 95 % workload. For the 50 % workload, we can observe that for all three protocols, the performance scales linearly with an increasing amount of threads. As discussed in section 5.3, the *append-requests* are the bottleneck for the 50 % workload. When viewing at the right plot of Figure 5.7 which shows the 95 % workload, we can see that for a lower amount of threads, CR outperforms CRAQ. We assume that the reason for this is the same as discussed in the section 5.3. For a lower amount of threads, it seems like that the *append-requests* are still the bottleneck. When looking at higher amounts of threads, the bottleneck of CR, due to the *Tail* handling all *read-requests*, becomes evident. It is not as scaleable as CRAQ, which scales linearly as for the 50 % workload. We, therefore, confirm the findings of Gavrielatos *et al.* [48] that CRAQ provides good thread scaleability. Through the concurrent design, *Ikaria* can leverage multi-core processors efficiently. The thread scaleability is, among other things (e.g. efficient access of shared data), dependent on how well the underlying hardware can perform concurrent writes and reads. As mentioned in subsection 2.1.2, the current PM hardware does not perform very well under concurrent access. This is why we propose to make use of multiple PM hardware modules instead of writing the log to a single PM stick (see Section subsection 3.2.3). By following this design, a good multi-thread performance for the log is to be expected.

Another important aspect of a shared log is how well the application performs with different *logEntry* sizes. Up until now, we measured the performance with a 256 B log

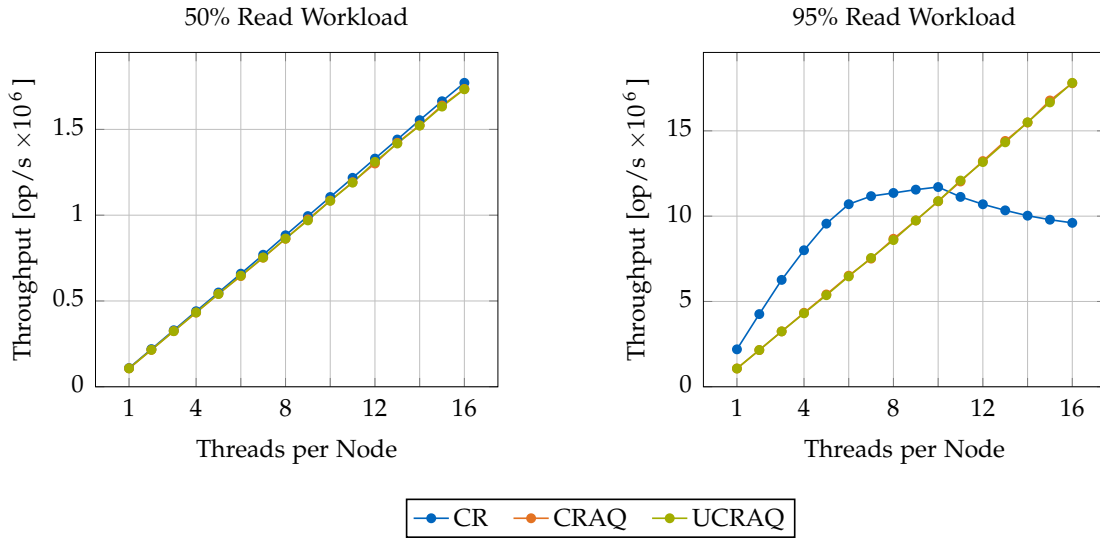


Figure 5.7: Performance of *Ikarria* for different multi-threaded configurations from one thread up to 16 threads on a 4-node chain.

entry size. In Figure 5.8 we compare the performance between values starting from cache line size 64 B, PM most performant value size 256 B up to the standard linux page size 4096 B. Due to hardware limitation of the DRAM we could only run the experiments for 4096 B for around 5 s and respectively the experiments for 1024 B and 2048 B. We, therefore, let these experiments run 12 times in total. Since the values varied not more than 20×10^3 op/s after removing outliers before calculating the average, we argue that our values are representative.

For the 50 % read-workload, the *append-requests* are the bottleneck as described before, and therefore all protocols perform nearly the same. As expected, we observe that the graphs decline drastically with bigger-sized `logEntries`. From 3×10^3 op/s for an entry size of 64 B down to 160×10^3 op/s for 4096 B. Since the entries have to be replicated on every node, a bigger entry size takes longer to persist than a smaller one. For the 95 % read-workload, we can see CRAQ/UCRAQ outperforming CR up to an entry size of 256 B. CRAQ/UCRAQ are able to process around 30×10^6 op/s for an entry size of 64 B whereas CR can't process more than 12×10^6 op/s which is CR's op/s bottleneck. Since the graph course for CRAQ/UCRAQ with 50 % workload, where the *append-requests* are the bottleneck, runs similarly to the 95 % workload plot, we assume that the *append-requests* are the bottleneck for this workload as well. CR and CRAQ/UCRAQ are only able to handle 1.5×10^6 op/s in total with an append share

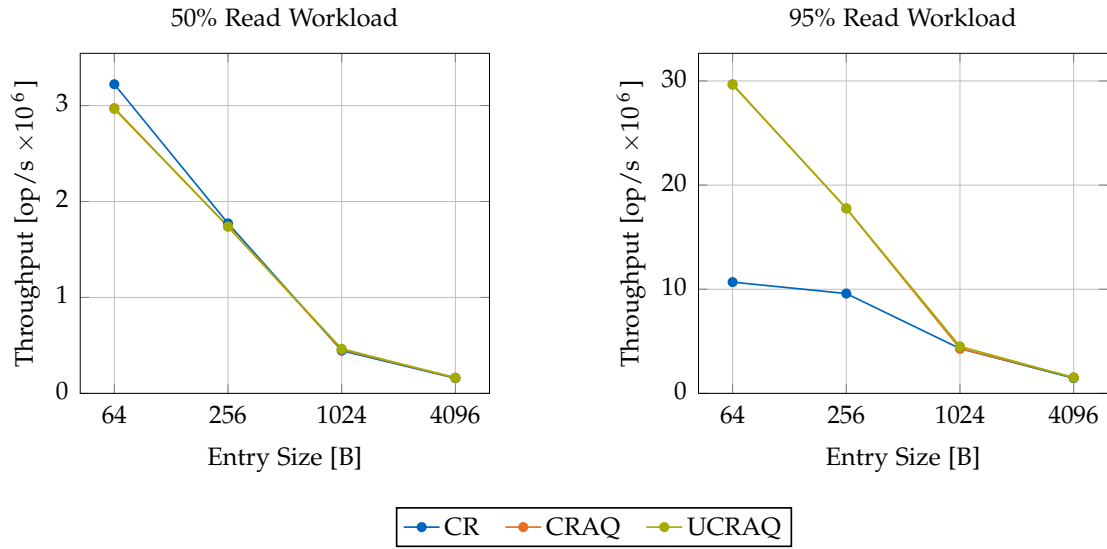


Figure 5.8: Evaluation of the scalability of Ikaria on a 4-node chain for different log entry sizes from cache line 64 B up to the standard Linux page cache 4096 B.

of 85×10^3 op/s for a log entry size of 4096 B. This is probably due to the *append-requests* and therefore the bottleneck for CR are the *append-requests* again and not the *read-requests* as for smaller entry sizes. As we mentioned in section 2.1 one advantage of PM is the byte-addressability and therefore the possibility to persist data which is smaller than 4096 B compared to SSDs where 4096 B is typically the smallest block size. Since we could observe that CRAQ/UCRAQ performed well for entry sizes smaller than 1024 B we argue that PM could perform great together with CRAQ/UCRAQ. This assumption should be validated in future work.

6 Related Work

In this section, we present an overview of previous work on shared logs. We focus on the general design and the workflow of write/read-requests for the specific shared logs since these are the most essential.

CORFU [6] has been presented in 2012 and is still the state-of-the-art shared log that provides strong consistency guarantees. CORFU has been designed as an intra-datacenter shared log. It builds upon the idea to decouple the ordering from the actual writing of the entries to storage. Instead of relying on storage servers, they directly use the network-attached clusters of flash units which is made possible through their client-centric design. Contrary to the server handling the mapping of log positions to flash pages, the clients do the mapping. The mapping, called projection, contains per log position multiple flash pages, depending on the desired replication factor. Through this mapping are the log accesses spread over all flash clusters. This, therefore, implements distributed wear-leveling since not one specific flash drive is accessed a lot and therefore has a shorter lifetime. To append a new entry, theoretically, the client could aggressively try to write to entries counting up until they find a free log entry. Since this would provoke heavy contention between the clients, CORFU offers as an optimization a dedicated sequencer, which returns for every request a dedicated *logPosition* (called *token*). After a client got the *logPosition* for its new entry, he then looks up in their local mapping which flash pages on which flash clusters correspond to the specific *logPosition*. For replication, CORFU uses a client-driven chain replication. Therefore the client now writes the new entry to every flash page, one after another. With the mapping, each client can perform *read-requests* by reading from the responsible flash drive. If a client received a *logPosition* but fails before actually writing the entry, the log will have a hole at this position. The paper suggests that other clients write junk values aggressively to empty log positions, which again provokes contention for the entry, especially when the failed client recovers. The maximum throughput of the log is, therefore, not dependent on the I/O bandwidth of a single storage server, but just how fast the sequencer can answer requests/distribute *logPosition*. One downside of CORFU is that in case of a flash cluster failure, the whole process pauses, the current projection has to be sealed, and every client has to recalculate the new mapping. CORFU has shifted the previous bottleneck of the storage server to the sequencer, which is now the limiting factor. [6]

Scalog [14] tries to solve some of the problems *CORFU* has. They use the same principle as *CORFU*, to decouple global ordering of all entries from actually persisting them. In contrast to *CORFU*, *Scalog* follows the idea of persisting all entries first and then assigning them a position. Therefore *Scalog* distinguishes between a data layer, which is responsible for storing and replicating the log, and an ordering layer, which assigns the *logPositions* to the individual entries. For enabling scalability, the data layer consists of multiple shards and each shard of multiple storage servers. When a client wants to append a new entry, it sends an *append-request* to a storage server in a shard of its choice, which enables data locality. The storage server stores the entry and replicates it via primary/backup replication in the specific shard to all other storage servers. It is essential to mention that each storage server has a dedicated segment for the log entries of the other storage servers. Each server periodically reports the number of persisted entries for all segments to the ordering layer. The ordering layer consists of a fault-tolerant replicated Paxos-based sequencer and aggregators as an optimization for collecting the persistent entry reports. It periodically sends a vector to every storage server in every shard, calculated from the shard reports. It includes the *logPositions* for the previous replicated entries for each server. The storage servers then assign the received positions and answer to the client. While offering good scalability, no reconfiguration timeout, and append speed, it does not support a general *read-request*. *Scalog* only offers a shard-specific *read-request*, where only *logPositions* can be read, which are replicated in this shard, and a *subscribe-request* where the client needs to keep connections to a storage server in every single shard. While this is sufficient for analytics applications, it could lead to problems when the use case has replicated state machines running on top of the log. The periodical waiting and reporting of the two layers adds up to higher latency, making *Scalog* not usable for applications requiring very low latency. [14]

Fuzzylog [15] is a globally distributed shared log which only offers partial-order instead of total-order over all entries. They assume that a system-wide total order is often unnecessary since updates to different chunks of data do not need to be totally ordered. *Fuzzylog* organizes updates in a directed acyclic graph (DAG) with happens-before relationships between the updates. Updates that originate in the same geographical region and access the same data are clustered together in totally ordered chains, and all chains are cumulated as a color. Therefore a color includes all updates for the same data, with updates from the same geographical regions totally ordered. The color is replicated at every client, but chains from elsewhere are only lazily synchronized between the locations and could therefore be stale. If a client wants to append a new log entry, he appends the update to the associated color's local, regional chain. Then he

adds the causality paths to the last seen update of every other chain in the same color. The updates in a chain are arranged in reverse order, therefore is the newest update the last entry. The whole DAG is ordered in a reverse topological sort A synchronize call is provided for updating a color, which gets all updates from the other chains, with the causality relationships between the different chains. The updates are then applied in the reverse topological sort order of the DAG. *Read-Requests* can just be done on the local copy, but as mentioned before, the returned data could be stale. Fuzzylog is, therefore, a good choice when a partial-order over the entries is sufficient for the application, and the log should be deployed globally. If this is the case, it offers linear scaling for throughput and availability in the case of network partitions. [15]

Delos [5, 9] has been presented in 2020 by Balakrishnan *et al.* Delos, in general, is a database that runs as a Replicated State Machine (RSM) on top of a shared log. Instead of having a specific shared log implementation that is directly accessed by the database, they virtualize the shared log. They propose to virtualize consensus by offering a virtualized shared log API called VirtualLog. The VirtualLog exposes an append-only and strongly consistent log to the clients. Internally, it maps the log entries to underlying Loglets. The Loglets can be distinct shared log implementation like for example the previous mentioned CORFU [6] or Scalog [14]. If a Loglet fails or another shared log implementation should be used, the VirtualLog seals the old Loglet and maps all new entries to the new Loglet. This makes it possible to foster new advancements in consensus research and swap out the underlying shared log implementation without any downtime. Therefore, the consensus is split up into the VirtualLog, which supports a reconfiguration capability for switching between the Loglets, and the Loglets that provide the ordering capability. [5, 9]

Tango [10] builds upon a shared log by offering replicated, in-memory data structures. For the data structures, it supports transactions as well. When a client wants to modify a Tango object in its memory, it sends a new log entry to the shared log with the desired changes. As soon as the entry is appended at the log, the client applies these changes to its in-memory data structure. Therefore, a Tango Object exists in the shared log as the history of changes and the client's memory. One advantage of the underlying shared log is that it can just replay the log entries when it comes back up in case of a client failure. Another advantage is that the single operations provide Strong Consistency guarantees. With Tango it is possible to build a general, highly available metadata service similar to Zookeeper [11]. Balakrishnan *et al.* show with their work how simple applications above a shared log can be implemented [10].

7 Conclusion

In the last chapter, we summarize our work and open up possibilities for future work. We also note our concerns and possible pitfalls regarding our evaluation.

7.1 Summary

We presented *Ikaria*, a shared log that is designed for read-heavy workloads with the recent advancements in networking and persistent memory. Therefore we make use of the asynchronous user-space networking library eRPC [47] for enabling fast and low-latency packet transmission. We consider the hardware characteristics of current available Persistent Memory modules and designed *Ikaria* in a way to leverage those at the best. One aspect where we consider the Persistent Memory hardware is the single-flush log design [59] for providing consistency instead of a two-flush design which proved less performant. For managing the Persistent Memory we make use of PMDK, which is a collection of libraries and tools to help developers leverage PM provided by Intel [29]. *Ikaria* supports Strong Consistency as well as Eventual Consistency by offering two different read operations. The two consistency levels open up greater flexibility for the clients in accessing log entries. As a replication protocol *Ikaria* incorporates CRAQ [56] which is based on Chain Replication but enables the possibility for reading locally on all nodes. This way *Ikaria* can run highly parallelized and offers high throughput for read-heavy workloads. Our implementation is based on a layered software architecture that makes it easy to, for example, exchange the used network library. In our evaluation, we show that CRAQ provides superior throughput for read-heavy workloads compared to Chain Replication. This proves the importance of a replication protocol that performs well when it is deployed with multiple threads. Furthermore, we demonstrate that CRAQ scales well with additional nodes compared to the original Chain Replication protocol. In conclusion, *Ikaria* is a modern shared log system that enables applications to leverage the performance gains of Persistent Memory and user-space networking.

7.2 Future Work

We have set a baseline with our design and implementation for *Ikaria*, but there are still some open tasks and questions which have to be addressed in the future. Some of those questions are regarding the evaluation of *Ikaria*. In the previous chapter 5 we could observe some behavior of CR and CRAQ which are hard to explain. Especially the question of why CR outperforms CRAQ in specific scenarios has to be further investigated. Another aspect we could not evaluate and has to be addressed by future work is how *Ikaria* performs with PM hardware. Due to our hardware limitations, we could only make assumptions about the performance, but evaluating our system with PM hardware is an important future task. The latency of the request types in *Ikaria* has not been evaluated in this work. Since the latency of the different request types is an important question, this should be evaluated in the future. As we make use of *eRPC* [47] in our implementation, which supports various transport layer protocols, another interesting question would be how *Ikaria* performs with RDMA instead of DPDK. Since there is an open discussion going on in the scientific community about RDMA and RPC, comparing these two with *Ikaria* could give valuable insights.

Besides the open work regarding the evaluation, *Ikaria* could be extended by some features in the future. Currently, *Ikaria* does not support any garbage collection for old log entries, so the log grows infinitely. Therefore, a *trim* command is needed for keeping the log at a manageable size. As we have seen in section 5.4 does tail reading of the log imply additional *state-requests* to the *Tail* node. To circumvent these additional messages, a *streaming-read* operation could be introduced. To avoid users trying to read new log entries aggressively, they can register themselves at a chain node. These registrations should be distributed over all chain nodes for good load balancing. Every time a chain node receives an acknowledgment message from the *Tail* for some newly committed log entry, it updates the state of the entry and sends a message to every client who registered itself. This request type could be a valuable extension since *Ikaria* is specially designed for read-heavy workloads.

7.3 Threats to Validity

In this section, we shortly want to state aspects we think could have influenced the benchmarks. As discussed in section 5.1 we are generating the workload in the application itself on the same machines. One problem hereby could be that generating many messages on the same servers where the application runs is that it limits the available resources for the application under test. Generated *append-requests* by the clients on the *Head* node can be directly processed and forwarded to the *Head's* successor.

Therefore there is not any added network latency for these messages compared to the *append-requests* from other nodes which have to send them to the *Head*. The same problem applies for generated *read-requests* on the tail node for CR. Another aspect to consider is other processes that run on the servers simultaneously while we perform the benchmarks. Since *Ikarria* is writing many data to DRAM, the benchmarks could have been influenced by other active processes which are accessing the memory.

List of Figures

2.1	RDMA Read/Write Data Flow	6
3.1	<i>Ikaria</i> System Overview	11
3.2	<i>Ikaria</i> Software Architecture (UML Component Diagram)	12
3.3	logEntry in-flight struct	13
3.4	<i>Append-request</i> Message Flow (UML Collaboration Diagram)	15
3.5	<i>Append-request</i> Message Flow (UML Sequence Diagram)	16
3.6	<i>Append-response</i> Message Flow (UML Sequence Diagram)	16
3.7	<i>Read-request</i> Message Flow (UML Collaboration Diagram)	18
3.8	<i>Read-request</i> Message Flow (UML Sequence Diagram)	19
3.9	logEntry struct	21
4.1	<i>Ikaria</i> Deployment (UML Deployment Diagram)	30
5.1	Evaluation Operations Append-only In-Flight Messages	34
5.2	Evaluation Operations Read-only In-Flight Messages	35
5.3	Evaluation Operations read/append-only workload	36
5.4	Evaluation Operations Chain Sizes	37
5.5	Evaluation Operations Read-Workload	39
5.6	Evaluation Operations Interval	40
5.7	Evaluation Operations Threads	42
5.8	Evaluation Operations Data Sizes	43

Bibliography

- [1] *Apache Kafka*. [Online]. Available: <https://kafka.apache.org/> (visited on 08/30/2021).
- [2] *Amazon Aurora: Design Considerations for High Throughput Cloud-native Relational Databases*. [Online]. Available: <https://www.amazon.science/publications/amazon-aurora-design-considerations-for-high-throughput-cloud-native-relational-databases> (visited on 11/08/2021).
- [3] *LogDevice · Distributed storage for sequential data*. [Online]. Available: <https://logdevice.io/index.html> (visited on 11/08/2021).
- [4] M. Van Steen and A Tanenbaum, “Distributed systems principles and paradigms,” *Network*, vol. 2, p. 28, 2002.
- [5] M. Balakrishnan, J. Flinn, C. Shen, M. Dharamshi, A. Jafri, X. Shi, S. Ghosh, H. Hassan, A. Sagar, R. Shi, J. Liu, F. Gruszczyński, X. Zhang, H. Hoang, A. Yossef, F. Richard, and Y. J. Song, “Virtual Consensus in Delos,” in *14th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 20)*, 2020, pp. 617–632, ISBN: 978-1-939133-19-9. [Online]. Available: <https://www.usenix.org/conference/osdi20/presentation/balakrishnan> (visited on 05/02/2021).
- [6] M. Balakrishnan, D. Malkhi, V. Prabhakaran, T. Wobblers, M. Wei, and J. D. Davis, “{CORFU}: A Shared Log Design for Flash Clusters,” in *9th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 12)*, 2012, pp. 1–14. [Online]. Available: <https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/balakrishnan> (visited on 04/04/2021).
- [7] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” in *Concurrency: The Works of Leslie Lamport*, New York, NY, USA: Association for Computing Machinery, Oct. 2019, pp. 179–196, ISBN: 978-1-4503-7270-1. [Online]. Available: <https://doi.org/10.1145/3335772.3335934> (visited on 08/30/2021).
- [8] *Apache BookKeeper™ - Home*. [Online]. Available: <https://bookkeeper.apache.org/> (visited on 08/30/2021).

- [9] M. Balakrishnan, C. Shen, A. Jafri, S. Mapara, D. Geraghty, J. Flinn, V. Venkat, I. Nedelchev, S. Ghosh, M. Dharamshi, J. Liu, F. Gruszczynski, J. Li, R. Tibrewal, A. Zaveri, R. Nagar, A. Yossef, F. Richard, and Y. J. Song, “Log-structured Protocols in Delos,” in *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles*, ser. SOSP '21, New York, NY, USA: Association for Computing Machinery, Oct. 2021, pp. 538–552, ISBN: 978-1-4503-8709-5. DOI: 10.1145/3477132.3483544.
- [10] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck, “Tango: Distributed data structures over a shared log,” in *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, ser. SOSP '13, New York, NY, USA: Association for Computing Machinery, Nov. 2013, pp. 325–340, ISBN: 978-1-4503-2388-8. DOI: 10.1145/2517349.2522732.
- [11] *Apache ZooKeeper*. [Online]. Available: <https://zookeeper.apache.org/> (visited on 08/26/2021).
- [12] *Pub/Sub for Streaming Analytics*. [Online]. Available: <https://cloud.google.com/pubsub> (visited on 11/09/2021).
- [13] *AlibabaMQ for Apache RocketMQ: Distributed Message Queue*. [Online]. Available: <https://www.alibabacloud.com/product/mq> (visited on 11/09/2021).
- [14] C. Ding, D. Chu, E. Zhao, X. Li, L. Alvisi, and R. van Renesse, “Scalog: Seamless Reconfiguration and Total Order in a Scalable Shared Log,” in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 325–338, ISBN: 978-1-939133-13-7. [Online]. Available: <https://www.usenix.org/conference/nsdi20/presentation/ding> (visited on 05/02/2021).
- [15] J. Lockerman, J. M. Faleiro, J. Kim, S. Sankaran, D. J. Abadi, J. Aspnes, S. Sen, and M. Balakrishnan, “The FuzzyLog: A Partially Ordered Shared Log,” in *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, 2018, pp. 357–372, ISBN: 978-1-939133-08-3. [Online]. Available: <https://www.usenix.org/conference/osdi18/presentation/lockerman> (visited on 04/29/2021).
- [16] *Intel® Optane™ Persistent Memory*. [Online]. Available: <https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html> (visited on 08/16/2021).
- [17] M. Bailleu, D. Giantsidi, V. Gavrielatos, D. L. Quoc, V. Nagarajan, and P. Bhatotia, “Avocado: A Secure In-Memory Distributed Storage System,” in *2021 {USENIX} Annual Technical Conference ({USENIX} {ATC} 21)*, 2021, pp. 65–79, ISBN: 978-1-

- 939133-23-6. [Online]. Available: <https://www.usenix.org/conference/atc21/presentation/bailleu> (visited on 08/16/2021).
- [18] Q. Cai, S. Chaudhary, M. Vuppalapati, J. Hwang, and R. Agarwal, "Understanding host network stack overheads," in *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, ser. SIGCOMM '21, New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 65–77, ISBN: 978-1-4503-8383-7. DOI: 10.1145/3452296.3472888.
- [19] A. Kalia, D. Andersen, and M. Kaminsky, "Challenges and solutions for fast remote persistent memory access," in *Proceedings of the 11th ACM Symposium on Cloud Computing*, ser. SoCC '20, New York, NY, USA: Association for Computing Machinery, Oct. 2020, pp. 105–119, ISBN: 978-1-4503-8137-6. DOI: 10.1145/3419111.3421294.
- [20] S. Scargall, "Introducing the Persistent Memory Development Kit," in *Programming Persistent Memory: A Comprehensive Guide for Developers*, S. Scargall, Ed., Berkeley, CA: Apress, 2020, pp. 63–72, ISBN: 978-1-4842-4932-1. DOI: 10.1007/978-1-4842-4932-1_5.
- [21] J. Yang, J. Izraelevitz, and S. Swanson, "FileMR: Rethinking {RDMA} Networking for Scalable Persistent Memory," in *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020, pp. 111–125, ISBN: 978-1-939133-13-7. [Online]. Available: <https://www.usenix.org/conference/nsdi20/presentation/yang> (visited on 03/12/2021).
- [22] S. Scargall, "Introduction to Persistent Memory Programming," in *Programming Persistent Memory: A Comprehensive Guide for Developers*, S. Scargall, Ed., Berkeley, CA: Apress, 2020, pp. 1–10, ISBN: 978-1-4842-4932-1. DOI: 10.1007/978-1-4842-4932-1_1.
- [23] S. Scargall, "Persistent Memory Architecture," in *Programming Persistent Memory: A Comprehensive Guide for Developers*, S. Scargall, Ed., Berkeley, CA: Apress, 2020, pp. 11–30, ISBN: 978-1-4842-4932-1. DOI: 10.1007/978-1-4842-4932-1_2.
- [24] I. O. p. m. r. a. g. t. i. D. w. t. n.-g. n. G. I. X. S. processors, T. W. O. T. W. H. B. E. M. A. I. f. D.-f. Cloud, databases, T. in Memory Analytics, and C. D. Networks, *Intel® Optane™ Persistent Memory Product Brief*. [Online]. Available: <https://www.intel.com/content/www/us/en/products/docs/memory-storage/optane-persistent-memory/optane-dc-persistent-memory-brief.html> (visited on 08/16/2021).
- [25] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson, "An Empirical Guide to the Behavior and Use of Scalable Persistent Memory," in *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*, 2020, pp. 169–182, ISBN:

- 978-1-939133-12-0. [Online]. Available: <https://www.usenix.org/conference/fast20/presentation/yang> (visited on 08/16/2021).
- [26] S. Scargall, "Operating System Support for Persistent Memory," in *Programming Persistent Memory: A Comprehensive Guide for Developers*, S. Scargall, Ed., Berkeley, CA: Apress, 2020, pp. 31–54, ISBN: 978-1-4842-4932-1. DOI: 10.1007/978-1-4842-4932-1_3.
- [27] DAX. [Online]. Available: <https://www.kernel.org/doc/Documentation/filesystems/dax.txt> (visited on 08/16/2021).
- [28] S. Scargall, "Fundamental Concepts of Persistent Memory Programming," in *Programming Persistent Memory: A Comprehensive Guide for Developers*, S. Scargall, Ed., Berkeley, CA: Apress, 2020, pp. 55–61, ISBN: 978-1-4842-4932-1. DOI: 10.1007/978-1-4842-4932-1_4.
- [29] *PMDK: Persistent Memory Development Kit*, Persistent Memory Programming, Aug. 2021. [Online]. Available: <https://github.com/pmem/pmdk> (visited on 08/16/2021).
- [30] *Pmem.io: Libpmem*. [Online]. Available: <https://pmem.io/pmdk/manpages/linux/master/libpmem/libpmem.7.html> (visited on 10/04/2021).
- [31] S. Scargall, "Libpmem: Low-Level Persistent Memory Support," in *Programming Persistent Memory: A Comprehensive Guide for Developers*, S. Scargall, Ed., Berkeley, CA: Apress, 2020, pp. 73–79, ISBN: 978-1-4842-4932-1. DOI: 10.1007/978-1-4842-4932-1_6.
- [32] *Pmem.io: Libpmemlog*. [Online]. Available: <https://pmem.io/pmdk/manpages/windows/master/libpmemlog/libpmemlog.7.html> (visited on 08/16/2021).
- [33] I. Moraru, D. G. Andersen, M. Kaminsky, N. Tolia, P. Ranganathan, and N. Binkert, "Consistent, durable, and safe memory management for byte-addressable non volatile main memory," in *Proceedings of the First ACM SIGOPS Conference on Timely Results in Operating Systems*, ser. TRIOS '13, New York, NY, USA: Association for Computing Machinery, Nov. 2013, pp. 1–17, ISBN: 978-1-4503-2463-2. DOI: 10.1145/2524211.2524216.
- [34] *Strata | Proceedings of the 26th Symposium on Operating Systems Principles*. [Online]. Available: <https://dl.acm.org/doi/10.1145/3132747.3132770> (visited on 08/16/2021).
- [35] *Quartz | Proceedings of the 16th Annual Middleware Conference*. [Online]. Available: <https://dl.acm.org/doi/10.1145/2814576.2814806> (visited on 08/16/2021).

- [36] J. Seo, W.-H. Kim, W. Baek, B. Nam, and S. H. Noh, "Failure-Atomic Slotted Paging for Persistent Memory," in *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '17, New York, NY, USA: Association for Computing Machinery, Apr. 2017, pp. 91–104, ISBN: 978-1-4503-4465-4. DOI: 10.1145/3037697.3037737.
- [37] *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. [Online]. Available: <https://www.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html> (visited on 08/21/2021).
- [38] Y. Zhu, H. Eran, D. Firestone, C. Guo, M. Lipshteyn, Y. Liron, J. Padhye, S. Raindel, M. H. Yahia, and M. Zhang, "Congestion Control for Large-Scale RDMA Deployments," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 523–536, Aug. 2015, ISSN: 0146-4833. DOI: 10.1145/2829988.2787484.
- [39] C. Guo, H. Wu, Z. Deng, G. Soni, J. Ye, J. Padhye, and M. Lipshteyn, "RDMA over Commodity Ethernet at Scale," in *Proceedings of the 2016 ACM SIGCOMM Conference*, ser. SIGCOMM '16, New York, NY, USA: Association for Computing Machinery, Aug. 2016, pp. 202–215, ISBN: 978-1-4503-4193-6. DOI: 10.1145/2934872.2934908.
- [40] Z. Istvan, D. Sidler, G. Alonso, and M. Vukolic, "Consensus in a Box: Inexpensive Coordination in Hardware," in *13th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 16)*, 2016, pp. 425–438, ISBN: 978-1-931971-29-4. [Online]. Available: <https://www.usenix.org/conference/nsdi16/technical-sessions/presentation/istvan> (visited on 08/13/2021).
- [41] A. Kalia, M. Kaminsky, and D. Andersen, "Datacenter RPCs can be General and Fast," in *16th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 19)*, 2019, pp. 1–16, ISBN: 978-1-931971-49-2. [Online]. Available: <https://www.usenix.org/conference/nsdi19/presentation/kalia> (visited on 05/16/2021).
- [42] *DPDK*. [Online]. Available: <https://www.dpdk.org/> (visited on 08/16/2021).
- [43] B. Brock, Y. Chen, J. Yan, J. D. Owens, A. Buluç, and K. Yelick, "RDMA vs. RPC for Implementing Distributed Data Structures," *arXiv:1910.02158 [cs]*, Oct. 2019. arXiv: 1910.02158 [cs]. [Online]. Available: <http://arxiv.org/abs/1910.02158> (visited on 05/15/2021).
- [44] *Persistent Memory Replication Over Traditional RDMA Part 1:...* [Online]. Available: <https://www.intel.com/content/www/us/en/developer/articles/technical/persistent-memory-replication-over-traditional-rdma-part-1-understanding-remote-persistent.html> (visited on 10/20/2021).

- [45] H. Bi and Z.-H. Wang, "DPDK-based Improvement of Packet Forwarding," *ITM Web of Conferences*, vol. 7, p. 01 009, Jan. 2016. DOI: 10.1051/itmconf/20160701009.
- [46] W. Zhu, P. Li, B. Luo, H. Xu, and Y. Zhang, "Research and Implementation of High Performance Traffic Processing Based on Intel DPDK," in *2018 9th International Symposium on Parallel Architectures, Algorithms and Programming (PAAP)*, Dec. 2018, pp. 62–68. DOI: 10.1109/PAAP.2018.00018.
- [47] *eRPC*. [Online]. Available: <https://erpc.io/> (visited on 08/16/2021).
- [48] V. Gavrielatos, A. Katsarakis, and V. Nagarajan, "Odyssey: The Impact of Modern Hardware on Strongly-Consistent Replication Protocols," in *Proceedings of the 16th European Conference on Computer Systems 2021*, Association for Computing Machinery (ACM), Jan. 2021. [Online]. Available: <https://www.research.ed.ac.uk/en/publications/odyssey-the-impact-of-modern-hardware-on-strongly-consistent-repl> (visited on 05/05/2021).
- [49] R. Van Renesse and F. Schneider, "Chain Replication for Supporting High Throughput and Availability,," Jan. 2004, pp. 91–104.
- [50] D. Ongaro and J. Ousterhout, "In Search of an Understandable Consensus Algorithm," in *2014 {USENIX} Annual Technical Conference ({USENIX} {ATC} 14)*, 2014, pp. 305–319, ISBN: 978-1-931971-10-2. [Online]. Available: <https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro> (visited on 09/23/2021).
- [51] A. D. Fekete and K. Ramamritham, "Consistency Models for Replicated Data," in *Replication: Theory and Practice*, ser. Lecture Notes in Computer Science, B. Charron-Bost, F. Pedone, and A. Schiper, Eds., Berlin, Heidelberg: Springer, 2010, pp. 1–17, ISBN: 978-3-642-11294-2. DOI: 10.1007/978-3-642-11294-2_1.
- [52] E. A. Brewer, "Towards robust distributed systems (abstract)," in *Proceedings of the Nineteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '00, New York, NY, USA: Association for Computing Machinery, Jul. 2000, p. 7, ISBN: 978-1-58113-183-3. DOI: 10.1145/343477.343502.
- [53] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, pp. 40–44, Jan. 2009, ISSN: 0001-0782. DOI: 10.1145/1435417.1435432.
- [54] *AWS | Amazon DynamoDB – NoSQL Online Datenbank Service*. [Online]. Available: <https://aws.amazon.com/de/dynamodb/> (visited on 09/23/2021).
- [55] P. A. Alsberg and J. D. Day, "A principle for resilient sharing of distributed resources," in *Proceedings of the 2nd International Conference on Software Engineering*, ser. ICSE '76, Washington, DC, USA: IEEE Computer Society Press, Oct. 1976, pp. 562–570.

- [56] J. Terrace and M. J. Freedman, "Object Storage on {CRAQ}: High-Throughput Chain Replication for Read-Mostly Workloads," in *2009 {USENIX} Annual Technical Conference ({USENIX} {ATC} 09)*, 2009. [Online]. Available: <https://www.usenix.org/conference/usenix-09/object-storage-craq-high-throughput-chain-replication-read-mostly-workloads> (visited on 05/05/2021).
- [57] M. Richards, *Software Architecture Patterns*. O'Reilly Media, Incorporated 1005 Gravenstein Highway North, Sebastopol, CA..., 2015, vol. 4.
- [58] *Storage Performance Development Kit*. [Online]. Available: <https://spdk.io/> (visited on 09/30/2021).
- [59] A. Van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper, "Persistent Memory I/O Primitives," in *Proceedings of the 15th International Workshop on Data Management on New Hardware*, ser. DaMoN'19, New York, NY, USA: Association for Computing Machinery, Jul. 2019, pp. 1–7, ISBN: 978-1-4503-6801-8. DOI: 10.1145/3329785.3329930.
- [60] R. Schiekofler, J. Behl, and T. Distler, "Agora: A Dependable High-Performance Coordination Service for Multi-cores," in *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, Jun. 2017, pp. 333–344. DOI: 10.1109/DSN.2017.23.
- [61] K. Walisch, *Libpopcnt*, Sep. 2021. [Online]. Available: <https://github.com/kimwalisch/libpopcnt> (visited on 10/01/2021).
- [62] *Templates - cppreference.com*. [Online]. Available: <https://en.cppreference.com/w/cpp/language/templates> (visited on 09/29/2021).
- [63] *Pmem.io: PMDK man page*. [Online]. Available: https://pmem.io/pmdk/manpages/linux/master/libpmem/pmem_memmove_persist.3 (visited on 10/22/2021).
- [64] *Std::popcount - cppreference.com*. [Online]. Available: <https://en.cppreference.com/w/cpp/numeric/popcount> (visited on 10/01/2021).
- [65] W. Muła, N. Kurz, and D. Lemire, "Faster Population Counts Using AVX2 Instructions," *The Computer Journal*, vol. 61, no. 1, pp. 111–120, Jan. 2018, ISSN: 0010-4620, 1460-2067. DOI: 10.1093/comjnl/bxx046. arXiv: 1611.07612.
- [66] *Std::atomic - cppreference.com*. [Online]. Available: <https://de.cppreference.com/w/cpp/atomic/atomic> (visited on 09/29/2021).
- [67] *Std::call_once - cppreference.com*. [Online]. Available: https://en.cppreference.com/w/cpp/thread/call_once (visited on 09/30/2021).
- [68] G. Marsaglia, "Xorshift RNGs," *Journal of Statistical Software*, vol. 08, Jan. 2003. DOI: 10.18637/jss.v008.i14.

Bibliography

- [69] *Rand(3) - Linux manual page*. [Online]. Available: <https://man7.org/linux/man-pages/man3/srand.3.html> (visited on 09/02/2021).
- [70] *3. Environment Abstraction Layer — Data Plane Development Kit 21.08.0 documentation*. [Online]. Available: http://webcache.googleusercontent.com/search?q=cache:aRHIF0jZu6AJ:https://doc.dpdk.org/guides/prog_guide/env_abstraction_layer.html&client=firefox-b-d&hl=de&gl=de&strip=1&vwsrc=0 (visited on 08/21/2021).