

Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Masterarbeit Nr. 26

State Machine Replication for Highly Available Service Networks

Jan-Hendrik Pauls

Course of Study:	Informatik
Examiner:	Prof. Dr. rer. nat. Dr. h. c. Kurt Rothermel
Supervisor:	Dr. rer. nat. Frank Dürr, M. Sc. Lukas Simon Probst
Commenced:	2015-04-01
Completed:	2015-09-30
CR-Classification:	C.2.2, C.2.4, C.4, D.4.4, D.4.5, D.4.7, H.3.4

Kurzfassung

Hohe Verfügbarkeitsanforderungen an vernetzte Dienste mit beliebiger Netzwerktopologie, wie sie aus dem Bereich der Verarbeitung komplexer Ereignisse als Operatorgraphen bekannt sind, werfen die Frage auf, ob die von klassischen Datenbank- und Client/Server-Anwendungen bekannten Replikationsmechanismen auch auf allgemeine vernetzte Dienste angewendet werden können. Hierbei bietet die Replikation von Zustandsmaschinen bekanntermaßen sowohl hohe Verfügbarkeit als auch starke Konsistenzgarantien, selbst wenn einer oder mehrere Knoten eines Dienstes ausfallen.

Insbesondere durch den Ausfall von Knoten, die nach einem Neustart ihren Zustand wiederherstellen, ist allerdings nicht klar, ob oder wie diese Konsistenzgarantien auf den vernetzten Dienst als Ganzes übertragen werden können. Hierfür beschreiben wir ein Protokoll, das die Verknüpfung replizierter Dienste zu einem konsistenten, semidynamischen Gesamtsystem ermöglicht. Das Protokoll baut dabei vor allem auf eine eindeutige Nummerierung der durch die Zustandsmaschine erzeugten Nachrichten sowie ein um "intelligente" Garantien erweitertes repliziertes Log auf. Ein formeller Beweis verifiziert die Korrektheit des Protokolls sowohl unter normalen Betriebsbedingungen als auch während der Wiederherstellung des Zustands. Schließlich gibt eine quantitative Evaluierung einen ersten Eindruck der möglichen Leistung des Protokolls.

Abstract

Service networks of arbitrary network topology are known from stream and complex event processing systems. As service networks are often required to be highly available, this raises the question, whether replication mechanism known from traditional database or client/server systems can be transferred to generic service networks. State machine replication is well-known to provide both, availability and strong consistency, even in the presence of node failures.

When nodes crash and recover, it is, however, not clear, whether, and in what way, these consistency guarantees can be transferred to the service network as a whole. To solve this, we propose a protocol that enables the interconnection of replicated services to a consistent, semi-dynamic service network. Our protocol is built upon a persistent enumeration system for messages produced by the state machine and upon a replicated log, which provides "intelligent" guarantees on the entries that are committed. A formal proof verifies the correctness of the protocol for normal operation as well as during recovery. We finally conduct a quantitative evaluation to give an impression of the possible performance of the protocol.

Contents

1. Introduction	9
2. Background	11
2.1. Service Networks	11
2.2. State Machine Replication for Highly Available Service Networks	14
2.3. Achieving Consistency for State Machine Replication	15
2.4. Leader Election	20
2.5. Related Work	20
3. System Model and Problem Statement	23
3.1. System Model	23
3.2. Problem Statement	24
4. Protocol and Abstract Implementation	29
4.1. Protocol	29
4.2. Basics and Notation	31
4.3. Static and Dynamic Information	32
4.4. Building Blocks	34
4.5. Operations	39
5. Concrete Implementation Using ZooKeeper	53
5.1. In Search of a Replicated Log	53
5.2. Employing ZooKeeper as Replicated Log	56
6. Proof	61
6.1. Preliminaries	61
6.2. Lemmata	63
6.3. Proof of the Normal Operation Properties	71
6.4. Proof of the Recovery Properties	73
7. Evaluation	77
7.1. Setup	77
7.2. Behavior under Load	77
7.3. Vertical Scaling	78
7.4. Horizontal Scaling	79

8. Conclusion and Outlook	81
A. A Simple Implementation of an “Intelligent” Replicated Log	83
Bibliography	85

List of Figures

3.1. Tier Layers	25
4.1. Chain of Tiers	29
4.2. Building Blocks of a Tier Component	35
4.3. Connection Establishment	43
4.4. Determining the Least Recent Snapshot to Keep	49
5.1. ZooKeeper Example Tree	56
5.2. The ZooKeeper Tree as a Replicated Log	58
7.1. Benchmarking Setup	78
7.2. Load Benchmark	79
7.3. Vertical Scaling	80
7.4. Horizontal Scaling	80

List of Algorithms

4.1. Bootstrapping from Initial State	40
4.2. Recovery from a Snapshot	41
4.3. Connection Establishment at the Producer Side	42
4.4. Connection Establishment at the Consumer Side	43
4.5. Connection Closing at the Consumer Side	43
4.6. Handling New Messages Received via one of the Inbound Connections	44
4.7. Handling New Entries in the Replicated Log \mathcal{L}	45
4.8. Managing Outgoing Messages that are Created by the State Machine	47
4.9. Taking a Snapshot s	48
4.10. Auxiliary Functions for Log Pruning	50
4.11. Log Pruning	50

1. Introduction

Interdependent service networks such as operator networks in stream or complex event processing systems, workflow systems or interconnected web-based microservices are supposed to be highly available and should work consistently even in the presence of machine or network failures. However, with the number of possibly failing components in such a service network, the total availability decreases. This can be circumvented by replicating each single service to multiple nodes. For stateful services, a well-known approach is state machine replication [Lam78; Sch90]. In an unreliable environment with node and network failures, replication raises the problem of consistency. While this problem has been solved for a single replicated service using consensus, replication or multicast protocols such as Paxos [Lam98], Viewstamped Replication [OL88], Zab [JRS11] or Raft [OO14a], it is not trivial to ensure that this also yields consistent behavior for other replicated services or the service network as a whole. For instance, when a machine of a replicated service crashes and recovers, it might not have replicated all messages which it sent before the crash or it might reproduce and resend messages which it has already sent before the failure.

In this work, we propose a protocol which allows the consistent interconnection of multiple so-called tiers, which each represent a single replicated service, to a service chain or service network. The network topology can be equivalent to an arbitrary graph, the services may comprise arbitrary state and even all nodes of a service may fail without harming the consistency properties. For availability and progress, we require a majority of nodes for each service to be available. Then, we offer availability with sequential consistency guarantees not only for a single service, but for the whole service network or each path in it, respectively.

Roadmap

This work is structured as follows. In the next chapter, we outline background information from the respective areas of the field of distributed systems and review and compare our approach with related work. In chapter three, we describe our system model and formulate the properties that our protocol and our implementation should fulfill. Then, we specify the protocol that each tier follows and describe an abstract, theoretical implementation in detail. Chapter five covers how we converted the abstract implementation given in chapter four to a concrete implementation using Java as programming language and ZooKeeper as replication protocol which provides an “intelligent” replicated log as well as a solution for

leader election. In chapter six, we formally verify that the abstract implementation we propose in chapter four actually fulfills the guarantees we stated in chapter two. The seventh chapter contains benchmarking results that give a rough impression of the throughput and latency of our concrete implementation as well as scaling behavior in two directions. We finally conclude with a discussion of our work and give an outlook of still open issues and possible future work.

2. Background

2.1. Service Networks

Service networks can be found in various kinds of distributed systems. Examples are a network of service providers as in a composite service (in terms of service-oriented architecture (SOA) applications [Pap03]) or an operator graph of a complex event or stream processing system [GK02].

In both cases, service networks are characterized by the *loose coupling* of its components which each provide a significant part of the whole service. With an increasing number of components, each of which may fail independently and each of whose unavailability often leads to the unavailability of the whole service network, the average availability declines [CDK05].

Furthermore, by the concept of loose coupling, one component is not supposed to have more knowledge than absolutely necessary about the other components. However, the order or number of messages sent out by one component might change the behavior of another component which depends on the first one.

These problems lead to the formulation of three well-known properties that not only the components of a service network, but also the service network itself is supposed to exhibit [Bre00; GL02]: availability, consistency and partition tolerance.

2.1.1. Availability

Availability means that all messages sent to a service must be processed. As remarked earlier [GL02], this definition seems to be weak as there is no time specified until which a message needs to be processed. In context with partition tolerance (which will be defined below), however, this means, that even when only one non-failing part of the component is reachable via the network, all messages received by this part need to be processed and the response must in general not be delayed until the partitioning is resolved. This also decouples availability from partition tolerance.

2.1.2. Replication

While availability can be increased by using more reliable or even redundant components such as redundant power supplies, the probably most common approach for achieving high availability in the presence of failing machines and a failing network is spacial or logical *replication*. Besides increasing availability, replication is also used to increase performance by distributing requests to multiple machines and to enable fault-tolerance against other than crash or network failures [CDK05]. There are two kinds of replication that can be distinguished, although they often occur together.

The first one, *data replication*, means that multiple copies of the same data are distributed to multiple, independent nodes. Then, depending on the consistency requirements, it is sufficient, that either one, several or a majority of the nodes is functional and/or reachable via the network, but some may have failed or be unreachable.

The second kind of replication is *computation replication*. In this case, the same program is run multiple times by several independent nodes. Then, again depending on the consistency requirements, only one, multiple or a majority of the computation results is sufficient and some nodes may fail to execute the program or be completely unavailable either due to a node or due to a network failure.

Besides the two kinds of replication with respect to what is replicated, there are also two different common ways how replication is implemented. With *active replication*, a data change or computation requests is executed at all (available) replicas, whereas *passive* or *primary-backup replication* processes the change or computation requests at one replica – the so-called *primary* – and then sends the resulting state or computation result to all other nodes.

As already mentioned, replication inherently introduces consistency requirements as the multiple copies or computation results need to be kept consistent. While the resulting consistency requirements may include quantitative objectives such as latency and throughput, the often most important objectives of consistency are (some kind of) order and integrity – both of which are qualitative objectives. This leads us to the next subsection.

2.1.3. Consistency

The second desired property is consistency. In this context, strong or sequential [Lam79] consistency is the property that all messages are processed in a total order which is the same for every single node of a replicated component. This means, even though the system is distributed, the order seems to be the same as if a single node had processed them.

While sequential consistency is one of the stronger consistency models, it does not guarantee any order of events with respect to when they occurred according to the wall-clock time. If this requirement is added, the even stronger consistency model is called linearizability [HW90]

or atomic consistency. As it is impossible to perfectly synchronize real-world clocks in an asynchronous system, with consistency we will mean sequential consistency hereafter.

2.1.4. Partition Tolerance

Partition tolerance, finally, means that even though the network connection between two or more nodes or parts of a component has failed and all messages sent via the connection are being dropped, the system remains functional and fulfills its specification.

2.1.5. The CAP Theorem

Unfortunately, Brewer [Bre00] hypothesized in 2000 that only two of these three properties can be fulfilled for a distributed system. While Brewer's conjecture has been criticized in recent years [Aba10], it has been proven in a limited¹, but rigorous way by Gilbert and Lynch [GL02], and is known as the CAP (consistency, availability, partition tolerance) theorem since then.

This means, the CAP theorem offers three a priori equal choices:

1. Consistency and partition tolerance, but no availability in case of a partition (CP). This is the classic choice of distributed database systems with ACID² properties and/or transactional semantics.
2. Availability and partition tolerance, but no (strong) consistency in case of a partition (AP). AP is chosen by modern NoSQL database systems, which only guarantee eventual instead of sequential consistency.
3. Consistency and availability, but no partition tolerance (CA). This last option is only feasible if the network will never partition. In case of globally distributed services where the system designer cannot change the properties of the network connecting the components' nodes, this is unrealistic and thus excluded as an alternative to choose.

For designing a system, the remaining choices are CP and AP, that is, whether to sacrifice consistency or availability in case of a network partition. Recent discussion [Aba10] has shown, that these two choices are in practice almost never symmetrical: While CP systems are only unavailable in case of a network partition, AP systems are often eventually consistent by

¹Above, we defined consistency to mean sequential consistency. However, Gilbert and Lynch only proved the CAP theorem for atomic consistency. This naturally poses the question whether one can have all three properties when the consistency is relaxed to sequential consistency. However, Bailis et al. [Bai+13] showed that with the same argumentation, the theorem also holds for sequentially consistent systems whose availability specification does not allow write operations to be delayed until a partition is resolved.

²Atomicity, Consistency, Isolation and Durability

design. A symmetric choice, in contrast, would be degrading consistency in case of a network partition.

Furthermore, we want to mention the concept of PACELC, which Abadi introduced in the same article: In case of a Partition, how to trade off Availability and Consistency; Else – this is, in normal operation – how to trade off Latency and Consistency.

2.2. State Machine Replication for Highly Available Service Networks

To achieve highly available service networks, we use a technique called *state machine replication*, which was introduced by Lamport [Lam78] and explained in depth by Schneider [Sch90]. State machine replication makes a single service highly available such that in consequence the whole network is supposed to become highly available, too.

2.2.1. State Machines

Formally speaking, a (deterministic) state machine consists of the following:

1. S , a set of states with s_0 as initial state.
2. I , an input alphabet, which is in our case the set of commands a service component accepts.
3. O , an output alphabet, which is in our case the set of messages or commands a service component may emit.
4. $\delta: S \times I \mapsto S$, a (deterministic) transition function which changes the state.
5. $\omega: S \times I \mapsto \mathcal{P}(O)$, a (deterministic) output function which emits one or more messages when a command is processed.

Schneider [Sch90] gives a similar definition of a state machine, consisting of *state variables* and *commands* which change the state variables and may generate output. State variables are constant until a command changes them. Commands are in general issued from another machine, called *client*, even though in special cases the machine executing the state machine could issue commands to itself, too.

2.2.2. State Machine Replication

The idea of state machine replication is to execute several copies of the same state machine on independent nodes or processors. Furthermore, all those processors execute the same sequence of instructions (i_0, i_1, \dots) with $i_k \in I$. When all state machines have the same initial state s_0 , it follows from the definition of a state machine that – given there is no error in the execution of the instructions – the state of all machines after every instruction will be the same.

So, to ensure a consistent state for all replicas, two things are important:

1. The *initial state* s_0 has to be the same.
2. The sequence of commands, (i_0, i_1, \dots) with $i_k \in I$, which is applied to the state machines, has to be the same for every replica.

The first issue, an identical initial state, can be solved easily by designing and initializing the state machines appropriately.

The second issue – a unique order of commands – is the tricky point. This is due to the fact that multiple clients are allowed to issue commands in parallel and the network does not give any guarantees about message delivery, such that it is neither clear if nor when nor in which order the commands arrive at the various replicas even if all commands are sent by only a single client.

2.3. Achieving Consistency for State Machine Replication

As consistency is a necessary property for state machine replication, the next question is, how to achieve it. For state machine replication, the three main approaches to guarantee sequential consistency are the application of either a consensus protocol, a dedicated replication protocol suitable for state machine replication or a multicast or broadcast mechanism with either total or primary order guarantees. While they are all slightly different in their perspective, consensus and suitable replication protocols directly fulfill the requirements for state machine replication. Furthermore, Chandra and Toueg [CT96] have shown that consensus and the total order broadcast are in fact equivalent³ for asynchronous systems with fail-stop or even so-called Byzantine failures.

In this section, we will describe the consensus problem with two important algorithms, a replication approach for state machine replication and total order/atomic multicast as well as the more advanced primary order multicast together with an algorithm for primary order multicast.

³Assuming an active replication scheme.

2.3.1. Consensus

The main idea of the consensus problem is how a set of distributed processes can achieve *agreement* [PSL80], that is they should agree on a common value which can be proposed by one or multiple processes and must not be changed after the processes decided to agree upon it. A well-known example is the Byzantine generals problem [LSP82].

More formally, there are three properties a consensus algorithm has to fulfill.

- *Termination*: Eventually, every correct process has decided upon a value.
- *Agreement*: If one correct process has decided upon a value, every other correct process has to decide upon the same value.
- *Integrity* (also known as *validity*): If all processes propose the same value, all correct processes have to agree upon this very value. Furthermore, the value the processes agree upon has to be proposed by some process.

While solving the consensus problem or achieving agreement is trivial for reliable systems and still solvable for synchronous systems which may fail [AW04; CDK05], Fischer, Lynch and Patterson [FLP85] showed in 1985 that it is impossible to solve the consensus problem in asynchronous systems in which only one process fails by stopping.

Luckily, there are approaches how to circumvent the rather strict assumptions of this proof like fault masking and randomization [CDK05]. The maybe most interesting approach is the one by Chandra and Toueg [CT96] in which *unreliable* failure detectors are used to distinguish arbitrarily slow from crashed processes. At the same time, the failure detectors may err to some degree without harming the correctness of the consensus protocol.

In the following, the two probably most widely used consensus algorithms are introduced briefly.

Paxos

Although Paxos has been unpublished for several years, it counts as probably the first and almost sure the most widely used consensus algorithm. After the general, but more complicated first version [Lam98], Lamport published a simplified version in 2001 [Lam01]. The implementation of Paxos is rather troublesome [CGR07], but powers important real-world services like Chubby [Bur06], a coordination service widely used at Google, Spanner [Cor+13], Google's globally distributed database, Amazon's Web Services [VS14] and many more [Wik15].

Raft

Although the consensus problem seemed to be solved by Paxos, the complexity of a correct implementation lead to a new popular consensus algorithm by Ongaro and Ousterhout: Raft [OO14a; OO14b]. Raft decouples leader election, log replication and safety and thus tries to enhance understandability as well as the correctness of implementations.

Raft basically works as follows. Firstly, a leader is elected by a majority of nodes. To simplify the leader election, Raft uses random timeouts such that in real-world scenarios there is one process that first asks to be elected. This leader needs to know all previously (possibly failed) rounds of the protocol to be elected, which renders an update of the leader's state redundant. Then, all nodes may forward requests to the leader which processes them in a unique order. First, the leader proposes the request's value, which could be for instance a state machine command. Then, all other nodes need to confirm this. As soon as a majority of nodes voted in favor of such a proposal, the leader may tell all nodes that the proposal is accepted. If then all proposals are ordered by the unique order given by the respective leader, one can see the notion of a replicated log to which entries are committed.

As every new leader has to re-propose all proposals it knows about, as every new leader is guaranteed to have the most recent information and the majority requirement leads to a quorum overlap, this guarantees that no proposal, that has been accepted, is ever forgotten.

The Raft protocol also specifies details on recovery, member changes and log compaction, we will not describe in more detail. Instead, we refer to the original publications [OO14a; OO14b].

2.3.2. Replication Protocols for State Machine Replication

A replication protocol is suitable for state machine replication if it can replicate a changing state such that all replicas see the same state changes in the exactly same order. This matches the above-mentioned sequential consistency criterion. Besides, we require the protocol to terminate and only apply effective⁴ state changes that have been issued by a client.

We do not specify whether the replication needs to be done actively or passively as both schemes are fine. Active replication schemes have to replicate the deterministic change operations while passive or primary-backup schemes only replicate the resulting states or state updates. While this enables non-deterministic change operations that are evaluated only by a primary and then turned into deterministic states or state updates, it also leads to the problem that multiple primaries may compete and issue conflicting state updates whose application may lead to inconsistent states [JS13]. This leads to the additional prefix order or primary order property

⁴The protocol may insert virtual, "no-op" change operations that do not change the state, but only serve as filler for the sequence to agree upon.

[JRS11] (see below) that passive replication schemes need to fulfill to meet the requirements of state machine replication.

Viewstamped Replication

Viewstamped Replication (VSR) [OL88; LC12] is a replication protocol that uses passive or primary-backup replication to replicate a stateful object. It has been developed at about the same time as Paxos and can be seen equivalent with respect to achieving state machine replication [LC12]. Above, it also guarantees primary order [RSS15]. Viewstamped Replication supports dynamic changes of the set of replicas and may not require stable storage if enough replicas are guaranteed to be available as failed replicas can then recover by obtaining the current state from other replicas. Furthermore, it has also been shown to support not only crash, but also Byzantine failures [Lis10].

2.3.3. Ordered Group Communication

The third alternative to achieve consistency is group communication. While group communication often also deals with the task of handling group membership and only enabling communication between actual members of the group, we focus on the task to send messages to multiple recipients in a particular order, which renders the so-called multicast equivalent to a network- or system-wide broadcast in which all components of a system receive the messages in the specified order.

While there are multiple order semantics, we focus on two of them. Both guarantee a specific order on all messages.

2.3.4. Total Order Multicast

Totally ordered multicast (or total order multicast; abbreviated TO multicast), which is also known as atomic multicast, provides a globally unique order of all messages. In particular, this means, that if any process in the system delivers a message m before it delivers m' , every correct process that delivers m' has to deliver m before m' . It is important to note, that this does *not* specify whether m has been sent before m' or not.

Total order multicast can be implemented in several ways [DSU04; CDK05]. The one which is very close to the implementations of consensus protocols assumes a sequencer, which can be external or elected among the communicating processes. All messages are then sent to the sequencer which determines the total order in which the messages are then delivered by all processes. In this case, the sequencer resembles the leader often used in consensus protocols.

To implement state machine replication, total order multicast can be used to agree upon the ordered sequence of commands the state machine executes to change its state. This equals the idea of active replication. If a passive replication scheme should be used, the primary order property has to be fulfilled as reasoned above.

2.3.5. Primary Order Multicast

Primary order multicast (abbreviated PO multicast) [JRS11] adds an additional condition to total order multicast and thus can be seen as refinement [RSS15] of total order multicast. There are two so-called primary order conditions where a primary is equivalent to a leader or sequencer.

- *Local primary order*: If one primary broadcasts a state update before another, all processes that deliver the updates have to deliver them in the order they were broadcast.
- *Global primary order*: If one primary broadcasts a state update u before a subsequent primary broadcasts an update u' , all processes that deliver the updates have to deliver u before u' .

Only if both properties are fulfilled, a multicast protocol is suitable for state machine replication if a passive replication scheme is to be used.

Zab

Zab [JRS11], which is ZooKeeper's [Hun+10] atomic broadcast protocol, lead to this explicit definition of the primary order property. Zab issues incremental state changes which are then broadcast to all replicas. Like Raft and VSR, it elects only replicas with the most recent information as primary. In addition to primary order, Zab also guarantees FIFO⁵ order for each client. However, while initially claimed [RJ08], it does not guarantee causal order, which is another message and event order guarantee, and the authors of Zab also show that primary order is strictly weaker [JRS11] than causal order.

2.3.6. Comparison

All three approaches, consensus algorithms, replication protocols and multicast schemes can be used to implement state machine replication. An extensive comparison including a refinement hierarchy has been published recently by van Renesse, Schiper and Schneider [RSS15]. Practically, all approaches have advantages and drawbacks and it depends on the

⁵First In First Out, that means, the sending order equals the order of delivery.

specific scenario which technique fits best. We will describe our needs in chapter 4 and our evaluation in chapter 5, where we also explain our choice.

2.4. Leader Election

While there is a considerable number of approaches to the leader election problem, we may limit ourselves to those that match the crash-recovery node failure model, which will be introduced in chapter 3. Furthermore, as leader election is not vital for the safety of our protocol, we only require eventual leader election. With the same argument, we refer to a recent publication by Larrea, Martín and Soraluze [LMS11] which includes a comprehensive overview and evaluation of previous work on the problem of leader election with a particular focus on the crash-recovery failure model.

2.5. Related Work

To the best of our knowledge, there is no previous work that deals with the question how multiple highly available tiers can consistently form a highly available generic service network. However, in this section, we will describe previous work that dealt with similar problems. The previous work can be divided into two categories, which are, firstly, consistent interactions between traditionally structured three-tier architectures or interactions using request/reply schemes and, secondly, fault-tolerant stream processing networks with stateful operators.

2.5.1. Consistent Interaction in Request/Reply or Three-tier Architectures

Mazouni, Garbinato and Guerraoui [MGG95a; MGG95b; Maz96] extensively dealt with the question how one replicated object can invoke a deterministic execution on another replicated object using proxies and filters to deduplicate invocations. While their notion of a replicated object is very similar to our notion of a tier, they only assume requests with replies. We, however, assume a chain or a generic network of services that may not only change the invoked object's state, but also produce messages subsequently sent to other service tiers when an execution is invoked.

This replicated invocation concept has been generalized to non-deterministic executions by Pleisch, Kupsys and Schiper [PKS03a; PKS03b], who also introduce the concept of orphan invocations that need to be dealt with in the context of transactions between replicated objects. They, too, cover only a request/reply scheme.

Frolund and Guerraoui [FG00; FG02] introduced the notion of e-transactions, that is, transactions for three-tier architectures that provide exactly-once semantics. Their publications

lead to a lot of work about how to design replicated three-tier architectures with transactional semantics, often using fault-tolerant CORBA as implementation framework. Many of the approaches are called multi-tier systems, but all that we found are equivalent to the a three-tier concept with separate clients, application servers and a database tier. Kolltveit [Kol04] provides an overview of those that provide exactly-once execution guarantees and high-availability and thus share the goals of our approach.

Zhao, Moser and Melliar-Smith [ZMMS02] proposed such an implementation for three-tier architectures using fault-tolerant CORBA with replicated gateways for inbound (client-side) and outbound (database-side) communication as well as replicated application servers. Their idea is similar to our approach, as they do deduplication within gateways. However, they use one layer of gateways between each of the three tiers, clients, application servers and databases, instead of viewing each application server as a separate and independent component which has to implement deduplication and other measures to ensure consistency.

Wu and Kemme [WK08] as well as Frolund and Guerraoui [FG01] provide theoretical concepts to validate exactly-once execution for replicated services. Wu and Kemme restrict their work to request/reply actions and passive replication, which does not fit our replication scheme-independent approach. While Frolund and Guerraoui's work is more restrictive than Wu and Kemme's work with respect to the actions – Frolund and Guerraoui assume either undoable or idempotent actions – their architecture and replication assumptions are more general and the guarantees they assume are equivalent to the execution guarantees that we want to provide among tiers. While we did not get to verify it in detail, we think that our proof shows that our implementation fulfills the criteria they postulated although we do not restrict the actions, that are executed by the tiers, in a similar way. However, this does not mean that their criterion can be extended as we do not assume transactional semantics as they do.

2.5.2. Fault-tolerant Stateful Stream Processing

The stream processing community follows a completely different approach. There, the topology of operators is arbitrary or at most limited to acyclic graphs and messages are sent in only one direction resulting in at most an acknowledgment, but no direct response to the sender.

Brito, Fetzer and Felber [BFF09a; BFF09b] use two categories of processing results, speculative and final ones, to minimize latency in replicated stream processing systems. While this is optimal for low-latency results, it requires very specific operator tiers and a rather complex protocol.

Martin et al. [Mar+11] utilize virtual synchrony to partially order and efficiently replicate operations, but assume that many operations are commutative.

Heinze et al. [Hei+13], finally, use a mixed active/passive replication protocol to make state machine-based operators in complex event processing systems fault-tolerant. However, their

2. Background

system seems to be limited to only one backup replica and they do not describe in detail how recovery and replay is made consistent.

3. System Model and Problem Statement

Essentially, all models are wrong, but some are useful.

(George E. P. Box)

3.1. System Model

Our system consists of several nodes, components, processes or machines¹ $\Pi = \{p_1, p_2, \dots\}$ which are connected via a computer network. This section describes how they are assumed to work and – maybe even more important – how they are allowed to fail without harming the guarantees provided by the protocol and implementation described in this work.

3.1.1. Nodes

Nodes are assumed to be independent computers which communicate by sending messages over the network. They can be segmented into functional units, called *tiers*².

Nodes progress at their own, unbounded speed independent of other nodes. They are equipped with two kinds of memory: volatile and non-volatile (also called persistent or stable) memory. Both are assumed to be readable and writable in an atomic manner, that is, data is read/written either completely or not at all.

Nodes fail only by crashing, which is also known as fail-stopping. This means they are allowed to stop processing at any arbitrary point of time, losing all data stored in their volatile memory. However, they may not – as assumed in the Byzantine failure model – process any other command or interact with their environment until they are restarted from some well-defined state.

Failed nodes may be restarted externally or by some kind of internal watchdog and thus are assumed to recover eventually. This leads to an overall node failure model which is also known as *crash-recovery*.

¹All four terms will be used interchangeably.

²We will denote a single tier with a lowercase t and a set of tiers with a capital T . An unannotated T , however, represents the set of all tiers.

Nodes are said to be *up* unless they crashed, then being *down* until they recover. Whether already the start or only the end of the recovery process is said to change the node's state from *down* to *up* depends on the mechanism chosen to achieve log replication. However, when a node has recovered and is fully functional again, we call it *functionally up*.

For progress of the consensus protocol, we assume that eventually enough nodes *per tier* are up sufficiently long (as defined in [CT96]) according to the respective definition of the replication protocol. Furthermore, we assume that at least one node *per tier* is functionally up sufficiently long as this is necessary to ensure progress across tiers.

3.1.2. Network

The network is a priori assumed to be asynchronous as defined in [CT96] and fair-lossy as defined in [ACT00].

Asynchronous means the nodes' clocks are not synchronized and the clocks may drift indefinitely. The delay of message transmission – as the time for nodes to execute a processing step – is not bounded either.

Fair-lossy means that no messages are created by the channel, messages are only duplicated a finite number of times and if a message is sent infinitely often, it will eventually be delivered infinitely often, too.

While both properties seem to make it rather difficult to provide the guarantees listed below, we can use unreliable failure detectors [CT96] and simple retransmission modules [BG05] or connection-oriented channels with automatic reconnection such as proposed by Guerraoui, Oliveira and Schiper [GOS98] to achieve consensus and eventual delivery nevertheless.

Note that these assumptions are only with respect to our protocol. The specific replicated log or in particular the leader election mechanism, which are both required later on, might have more demanding requirements on the network.

Besides these synchronicity and transmission properties, we assume that there is some kind of discovery and addressing mechanism such that all nodes can communicate with each other using only addresses and ports.

3.2. Problem Statement

Using state machine replication in chained or networked services has not yet been thoroughly examined. For tiers which employ state machine replication as a mean to ensure consistency in case of a machine or network failure, it is not trivial to link them while still guaranteeing consistency across tiers.

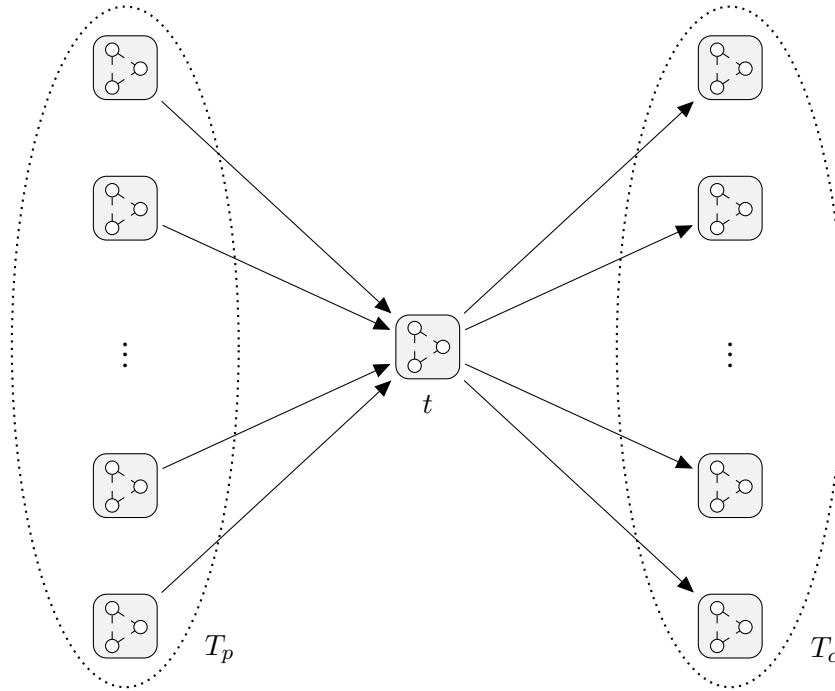


Figure 3.1.: Each tier t consumes messages from a set of tiers $t_p \in T_p$, applies them to its internal state machine. The internal state machine may not only change its state, but also produce messages, which are then delivered by a semi-dynamic set of consuming tiers $t_c \in T_c$.

3.2.1. Preliminaries

As it has been shown that state machine replication can be used to make a group of replicated state machines behave identically with respect to the messages they receive and send, this yields the following guarantees for each tier, which will be derived from the properties of state machine replication in chapter 6.

Lemma 3.1 *For each tier t , the set of incoming messages IM^t consumed by the state machine replicated within tier t as well as the strict total order $<_v^t$, making IM^t a totally ordered set, is identical for all tier components $t_i^t \in t$.* \square

Corollary 3.1 *For each tier t , the set of outgoing messages OM^t produced by the state machine replicated within tier t as well as the strict total order $<^t$, making OM^t a totally ordered set, is identical for all tier components $t_i^t \in t$.* \square

Based on these guarantees, we can propose consistency guarantees at a per-tier abstraction without specifying which tier component has to contribute which exact part. This leads to the model depicted in Figure 3.1: Each tier $t \in T$ is connected with a static set T_p^t of tiers it receives

messages from. These tiers are also called *producing*, *sending* or *preceding tiers*. Furthermore, it is connected to a *semi-dynamic* set $T_c^t(m)$ of tiers which in turn receive messages from tier t . This set is called *consuming*, *receiving*, *delivering* or *succeeding tiers*. $T_c^t(m)$ is semi-dynamic in the effect that it may grow with the number of produced messages. However, if some tier t_c ever joined $T_c^t(m)$ by delivering some message $m = m_{\text{init}}$, it may not leave $T_c^t(m)$ anymore. This leads to the following, semi-dynamic definition of $T_c^t(m)$:

$$T_c^t(m) := \{t \in T : t \text{ delivered } m_{\text{init}} : m \geq m_{\text{init}}\}$$

In the following, if we consider one tier with its related producing and consuming tiers, we may simplify the notation to T_p and $T_c(m)$, $t_i \in t$, IM and OM as well as $<_i$ and $<$. Furthermore, we will say some tier t produces or sends a message even though this is in fact done by the state machine replicated within tier t . Similarly, we say some tier t consumes, receives or delivers a message if it is in fact delivered to the state machine replicated within tier t .

3.2.2. Normal Operation

Having set this context, we propose the following consistency properties for normal operation. Normal operation is defined by the time until a node crashes or after it recovered consistently (which will be specified below).

Property 3.1: Suffix Validity

If tier t produces some message $m' > m$ after some other message m and some tier $t_c \in T_c(m) \subseteq T_c(m')$ delivered m , t_c will eventually deliver m' .

Property 3.2: Agreement

If some tier $t_c \in T_c(m)$ delivers a message m , then every tier $t'_c \in T_c(m)$ will deliver m .

Property 3.3: Total Production Order

If some tier t produces a message $m' > m$ after some other message m , every tier $t_c \in T_c(m')$ that delivers m' has to deliver m before it may deliver m' . Furthermore, every message m is delivered only once by each tier $t_c \in T_c(m)$. This is equivalent to $m' >^t m \Rightarrow m' >_i^{t_c} m$.

Property 3.4: Integrity

Every message m that is delivered by tier t is produced by exactly one preceding tier t_p .

Property 3.5: Initial Registration

There is at least one tier $t_c \in T_c(m_0)$ that delivers the very first message m_0 produced by t .

Less formally spoken, suffix validity implies that every consuming tier eventually delivers all produced messages from some initial message on. Agreement defines that all consuming tiers that receive some suffix of the produced messages have some point, which is defined by the

initial message of the last tier joining the respective T_c , from which on they receive the same set of messages from tier t . Total production order propagates the production order to the next tier, relaxing it only as far as necessary to allow messages from other tiers to merge in. Integrity ensures that there is no message creation by the channel or the rest of the implementation. Initial registration, finally, gives a well-defined fallback option for chains that yields validity not only for the suffix, but for all messages produced by a preceding tier in the chain.

By induction, the very same properties then hold for the whole chain or, in service networks, for each path through the network.

Note that we do not guarantee some kind of total order for messages that are delivered from two separate tiers to two other, separate tiers as this is only possible by knowing or influencing all involved sending or receiving tiers, which might not always be possible or desired in a real-world context. We also do not guarantee causal order, which might be interesting in some use cases, as we found no way to do so without restricting either the other guarantees or the topology of the network.

The restriction to the suffix is necessary due to the dynamic nature of the network. If the network is static and a tier t knows all succeeding tiers $t_c \in T_c$, it is an easy exercise to adapt the sending and log pruning implementation such that all those guarantees hold from the very first message on. As the set of succeeding tiers is not assumed to be known or even fix at any point of time, one has to decide between the ability to prune the log and only guarantee the above properties for the suffix or guarantee them from the very first message on, but not prune the log at all as each tier has to keep all messages forever³.

3.2.3. Recovery

The above guarantees assume normal operation. For recovery, we have to relax some of the properties in a way that – by the rest of the guaranteed properties – has only an effect for the directly succeeding tiers, but is caught there and does not propagate to further successors down the chain or path.

Property 3.6: Repeating Suffix Recovery

If some tier component $t_i \in t$ recovers from some initial state or snapshot as it crashed before processing some message m_{crash} , the ordered set of messages redelivered at its state machine replica during recovery is an ordered suffix of the messages delivered by t until t_i crashed, $\{m : m \in IM \wedge m <_i m_{\text{crash}}\}$, ordered by the same total order $<_i$ these messages were delivered by before the crash.

³The idea to start over from the initial state and request all missing log entries from the preceding tier – which then might have to start over from initial state, too – might seem compelling. By induction, however, one can easily prove that this only implies that then the first member or source of a chain or path is unable to prune its log.

Property 3.7: Message Identity

If some tier component $t_i \in t$ produces some message m' during recovery from some initial state or snapshot and this message is identical to another message m produced before it crashed – in particular identical regarding some identifier field – the production and delivery of m' and m is defined to be equivalent as they have the same effect on consuming tiers.

The repeating prefix property relaxes the total production order property in a way that allows a well-defined redelivery of messages during recovery. While the redelivery certainly may cause messages to be produced multiple times, the well-definedness allows succeeding tiers to deduplicate them easily and hence deliver them only once. This is specified in the message identity property, which forbids succeeding tiers that already delivered m to deliver m' , too, and allows succeeding tiers that did not yet deliver m to do this by delivering m' . Note that the equivalence of the production of m and m' automatically preserves the total order that would have been generated during normal operation.

4. Protocol and Abstract Implementation

This chapter describes the fundamental protocol and an abstract implementation thereof that is independent of the software framework, programming language or the tools used for networking, the replicated log or the leader election algorithm. In the next chapter, we will describe how we converted this abstract to a concrete, Java- and ZooKeeper-based implementation. However, we will use this simpler, abstract description to prove the correctness of our implementation in chapter 6.

4.1. Protocol

This section describes the protocol according to which the components of a tier behave and communicate with components of other tiers as well as with sources and sinks.

4.1.1. Components and Requirements

The protocol requires at least one tier. We assume that each tier runs exactly one state machine. This is only a logical separation as multiple instances of the protocol can run in parallel on the same physical node using different network addresses.

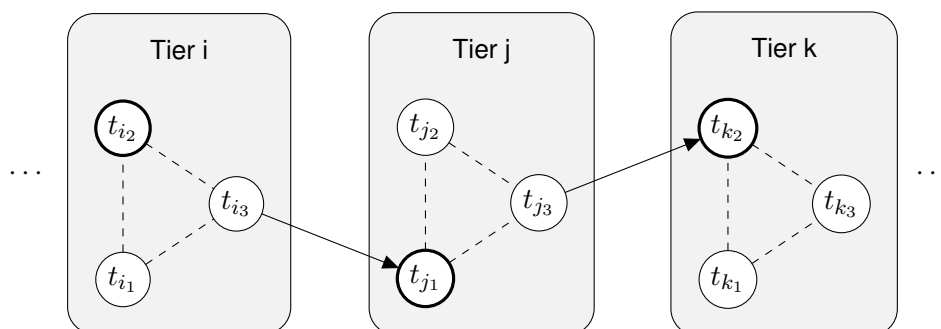


Figure 4.1.: A typical chain of tiers. Each tier consists of several nodes which are kept consistent with a replicated log. One of the nodes in each tier is elected as leader and only this leader node is allowed to establish connections to the preceding tiers through which the tiers then are able to exchange messages.

Each tier is held consistent by a replicated log. Furthermore, a leader election protocol ensures that, eventually, only one component per tier considers itself the leader of the tier. While the replicated log is necessary for the safety of the protocol, the leader election algorithm only improves its performance. Besides those two precast communication protocols, the tier components of one tier do not communicate among each other.

However, the tier components do interact with tier components of other tiers as well as sources and sinks. They do so according to the network model introduced in the previous chapter. The protocol does not distinguish tier components which fully implement the protocol from other network nodes that only produce messages, so-called *sources*, or only consume messages, so-called *sinks*. Therefore, we only specify the behavior of a tier component and it is up to the implementer to ensure that sources and sinks behave accordingly to achieve the same guarantees that also hold for ordinary tier components. For instance, if a source issues the same request twice using different identifiers, which may easily happen during incautious recovery, the tier not only can, but has to treat the requests as two independent entities.

4.1.2. Specification

Each tier component of a tier participating in the service network has to exhibit the following traits.

Connection Establishment

Each tier component has to offer a server interface to which a consuming tier can establish a connection. This server interface is to be provided as soon as a tier component is functionally up. Then, it is assumed that only one tier component per tier should establish a connection to some component of each producing tier. However, the consistency properties must not be compromised by more than one simultaneous connection per tier or even per tier component. It is assumed that a leader election protocol is used to achieve this eventually.

Input Consistency

Once a component of tier t has received some message m_{init} from a component of a producing tier t_p , it has to receive all subsequently produced messages $m \geq m_{\text{init}}$ which are sent in the order they are produced by the state machine inside t_p . Furthermore, the components of tier t will never receive any message that has been produced before m_{init} .

Acknowledgment Consistency

Once a component of tier t acknowledges the reception of a message from a preceding tier t_p , the message is replicated within t . Thereafter, no component of tier t_p will have to send this message ever again to tier t although it may do so.

Output Consistency

If some component of tier t receives a message m_i from a producing tier t_p , the components of tier t_p ensure that this message is sent to all consuming tiers t' that received some message $m \leq m_i$ when they establish a connection to t_p unless some component within t' already acknowledged m_i .

Total Production Order

A producing tier t_p has to specify the production order of the messages it sends by an identifier, id , that is unique per tier and consistently assigned for all tier components within this tier.

4.2. Basics and Notation

This section introduces the basic terms, presumptions and details of our notation used to formulate the abstract implementation.

4.2.1. Messages

Each message m consists of the following three things: A message id which must be unique per tier, a sender id , which is the id of the tier that produced m , and the actual content, which is as general as an array of bytes:

$$m_{id} = \{id = id(m), senderId = senderId(m), content = content(m)\}$$

Furthermore, each message includes a boolean $init$ flag, $init(m)$, which is false by default. This flag indicates the very first message a consuming tier should process.

4.2.2. Predefined and User-defined Functions

In the notation, we distinguish between predefined functions that are provided either by the operating system, the networking layer or more specific components like the replicated log or the leader election mechanism. A predefined FUNCTION will be underlined while a user-defined FUNCTION will be undecorated. If no arguments are passed to a function or they are irrelevant, we might omit the parentheses, which otherwise frame them.

4.2.3. Object Notation and Auxiliary Entries

For objects like a snapshot s , we use the notation $s:x$, when we access some field x of object s . A similar notation is used for auxiliary entries, which will be introduced later. There, an entry like $\text{minACK}:x$ or $\text{minSnap}:x:y$ indicates that this is an auxiliary entry of the respective type.

4.3. Static and Dynamic Information

This section describes the static and dynamic information given by the environment or stored in memory during runtime respectively.

4.3.1. Static Information

Besides implementation specific information like storage paths and possibly necessary address/port details for the replicated log, each tier has the following static information. Static means that this information will not change once the tier is started and might only change when the tier is offline.

- Information about the tier t :
 - The tier id, $\text{tierId}(t)$, an id which uniquely identifies the tier among all tiers in T .
 - Source information which consists of a set of $(\text{tierId}, \{\text{address:port}, \text{address:port}, \dots\})$ tuples, where the addresses and ports belong to the tier components of a preceding tier t_p which is identified by the id tierId .
- Information about the tier component t_i :
 - The component id, $\text{componentId}(t_i)$, an id which uniquely identifies the tier component within its tier.
 - The port to which the tier component's server should be bound to.

While tier id, source information and component id cannot be changed easily without hurting the consistency guarantees of the protocol, changing the port information could even be done online when all involved tiers and tier components are taken offline or at least their network interfaces do not establish any new connections or try to reconnect until all information is updated at all involved tiers.

4.3.2. Volatile and Non-volatile Variables

While static information is considered as constant and given at the begin of the execution, there are also dynamic variables that may change over time. These can be stored in volatile or non-volatile memory.

In the replicated log, which is initially empty and is assumed to recover automatically to the most recent state, the following variables are stored globally per tier t . As the log is assumed to recover completely after a crash, we see this as non-volatile memory.

- For each tier $t_p \in T_p$ that this tier t is consuming messages from:
A set $\mathcal{L}(t_p)$ of messages that are already received. This set can be viewed as ordered set, not only as the messages' ids provide a strict total complete order per tier, but also as the commit order of the log implies this. In particular for the superset of all $\mathcal{L}(t_p)$, \mathcal{L} , we always mean the order implied by the log.
- For each tier $t_c \in T_c$ that consumes messages produced by t 's state machine:
The smallest id $\text{minACK}(t_c)$ of all unacknowledged messages. This means all messages up to $\text{minACK}(t_c) - 1$ are acknowledged by t_c which – by acknowledgment consistency – implies that these messages have been successfully committed to the replicated log at t_c .
- For each tier component $t_i \in t$:
The largest log entry id $\text{minSnap}(t_i)$ that tier component t_i is currently able to recover from. This means that all entries up to this value can be safely removed from the log without compromising the ability to recover consistently.

Note that the last information is only necessary if the tier components take snapshots individually (in contrast to collective snapshotting algorithms) and thus use these log entries to convey information about their recoverability.

Besides the information stored in the log, we use snapshots that contain the state machine's state, but also additional information necessary for consistent numbering of outgoing messages or to enforce a proper order for all entries before they are applied to the state machine. These, too, are store in persistent memory and will be available after a crash failure.

In its volatile memory, each tier component stores the following variables:

- $smState$, the current state of the state machine.

- *lastLogId*, the log-specific id of the last entry applied to the state machine.
- *nextMessageId*, the id to be assigned to the next outgoing message produced by the state machine.
- *allMessages*, a set of messages generated by the state machine. This set at least contains all messages from $\min_{t_c \in T_c}(\text{minACK}(t_c))$ on, but might reach further back until the log is pruned. The set can be viewed as ordered set as the produced messages' ids provide a strict total complete order.
- *unAcknowledgedMessages*, a map that stores for each consuming tier $t_c \in T_c$ the set of message ids that are not yet acknowledged by t_c .
- *unhandledEntries*, a set of entries needed during recovery to manually apply all entries committed to the log since the snapshot the tier component is recovering from. This is an ordered set using the log id of the entries as strict total order.
- *lastProcessedMessage*, a map that – for each producing tier $t_p \in T_p$ – stores the id of the last message from t_p that has been applied to the state machine.
- *unprocessedMessages*, a map that stores the set of messages that are already committed to the log, but must not yet be applied to the state machine to enforce application in order of the message id. Hence, this map contains an ordered set for each producing tier $t_p \in T_p$ with the message id as the criterion to provide a strict total order.
- *mode*, a auxiliary variable that indicates whether the tier component is still recovering or already functionally up and operates normally.

lastProcessedMessage and *unprocessedMessages* are only necessary in case the replicated log does not enforce that messages are committed in the order of their message ids.

We also introduce local variables whose meanings become obvious from the surrounding algorithms.

4.4. Building Blocks

This section describes the modules or building blocks a tier component is built of. The building blocks and how they interact is depicted in Figure 4.2.

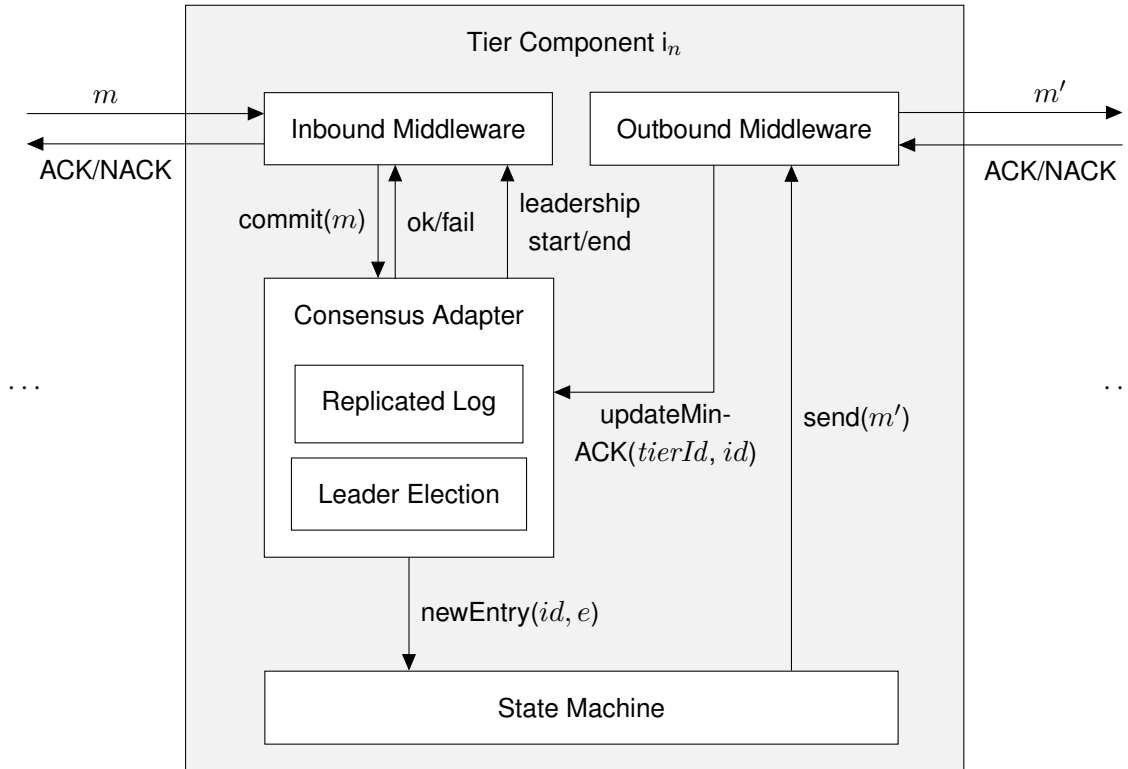


Figure 4.2.: The building blocks of a tier component.

Inbound connections are established and managed by the inbound middleware, which is only active if the tier component is elected as leader within its tier. The inbound middleware can commit messages to the consensus adapter, which may or may not confirm this.

In case there is a new entry, the consensus adapter notifies the state machine which then has to handle the entry. The consensus adapter also notifies the inbound middleware about leadership start/end so it can establish/cease connections to preceding tiers.

As the state machine handles new entries, it may produce messages. These are then sent to succeeding tiers, which is done via the outbound middleware.

The outbound middleware assigns unique ids to the messages produced by the state machine, handles outgoing connections from consuming tiers and manages the (replicated) acknowledgment state for each of the succeeding tiers.

4.4.1. The State Machine

Besides the static configuration information, the user can also specify the state machine. In this work, the state machine has a user-definable initial state, $smInitState$, can change its state as a reaction of incoming messages and can produce a finite number of outgoing messages when it reacts to an incoming messages. For simplicity, it is assumed that the state machine is deterministic. However – as already proposed by Liskov [LC12] –, this could be changed with ease by evaluating all non-deterministic functions that are invoked when a message is processed before the replication of a message and make the leader commit this additional information alongside the actual message or entry.

The state machine provides the $\text{APPLYENTRY}(id, e)$ function which is called whenever a new entry e with log id id has been successfully committed to the encapsulated log and should be handed over to the specified state machine code.

4.4.2. The Replicated Log \mathcal{L}

The replicated log \mathcal{L} – also called stable log or plainly log – is encapsulated by the consensus adapter. It offers read access to all replicas, but it is not guaranteed to see the most recent information. However, the information is only stale and not inconsistent in the way that some committed entry might be replaced by some other value. Read access is denoted by $\mathcal{L}(x)$ when a value x is to be read. To simplify the notation when tier-specific or auxiliary entries (introduced below) are read, we introduce the notations $\mathcal{L}(\text{tierId}(\cdot))$, $e \in \mathcal{L}$, $\mathcal{L}(\text{minACK}(\cdot))$ and $\mathcal{L}(\text{minSnap}(\cdot))$. If multiple entries exist for one notation, which is the case for the auxiliary entries, the log will always return the most recent value visible to the replica.

Furthermore, the replicated log offers a write function which we call $\text{COMMIT}(x)$. The commit function returns a failure unless either the value or message x has been successfully replicated or it is already present in the log – if this semantic is supported by the log. In general, it is possible to modify the replication mechanism. However, we will always assume a quorum replication, such as in Raft or zab, in which the leader or sequencer has the most recent information or most complete view of the log before it commits additional entries to the log.

Besides “normal” commit operations we also assume some kind of conditional commit $\text{COMMITIF}(x, cond)$ that may only succeed if the condition $cond$ is fulfilled. The condition is to be evaluated such that it holds for all entries that are committed up to entry x .

We just assumed a deduplication feature as well as some kind of conditional commit. While we describe in chapter 5 how we can easily implement these features for our ZooKeeper-based log, this might not be trivial for all replicated logs. Therefore, we propose a wrapper that enables these features for other, commit-only replicated logs in appendix A. This wrapper could be part of the consensus adapter in other implementations.

The fourth way to interact with the environment is that the log invokes the state machine's `NEWENTRY(id, e)` function whenever a new entry is successfully committed to the log. It is crucial that these invocations happen in the order of the log entries, i.e. the state machine is invoked with entry *i* if and only if the last invocation was with entry *i* - 1.

The replicated log may optionally enforce a specific order of incoming messages by enhancing the procedures according to which the leader decides which message to append to the log and which to discard. This *enforced order* semantics means that if for one tier t_p from which messages are received, message m_i is not yet committed, message m_{i+1} – and by induction all subsequent messages – must not be committed either, but their commit is either deferred (without returning the commit call) or returns with failure. If this semantic is available, some features of the implementation become dispensable and are marked as such. We also describe how to implement this for a simple, commit-only replicated log in appendix A.

As we also specify log compaction, the log provides two functions to accomplish that. The more general `PRUNELOG` function only prunes the normal, message-based log entries, keeping at least one entry for each tier the tier received messages from. The `PRUNEAUXILIARYLOG` function prunes all auxiliary entries, but the most recent one per tier or tier component, respectively. We will see in chapter 5 that pruning of auxiliary entries is not necessary for some log implementations.

4.4.3. Leader Election

As it is sufficient that only one tier component connects to the tiers from which messages should be received, we employ a leader election mechanism to determine which tier component does this at each point of time. As multiple parallel connections do no harm, we have very low requirements on the leader election algorithm considering ending the leadership. What we only require is that eventually, a functionally up process is elected. We do *not* require the other processes to know which of the other processes is leader. They only need to know their own status. A very simple implementation would be to adopt the leader the log replication algorithm elects as leader or sequencer.

The leader election mechanism provides callbacks, `LEADERSHIPSTART` and `LEADERSHIPEND`. They are automatically invoked as soon as the respective tier component is elected as leader or its leadership ended. It is left to the implementer to correctly set up the leader election mechanism.

4.4.4. The Consensus Adapter

The consensus adapter encapsulates the replicated log as well as the leader election mechanism. It is responsible for correct bootstrapping of both and provides a simple interface for both, replicated log and leader election.

4.4.5. The Inbound Middleware

The task of the inbound middleware is to establish connections with the tiers $t_p \in T_p$ from which messages should be received. It has to do this as soon as the tier component it belongs to is elected as a leader via the leader election mechanism. Furthermore, it has to manage reconnects in case the connection is disrupted or closed. It is assumed that the inbound middleware iterates over all tier components t_{p_i} of each tier t_p while reconnecting such that an ongoing unavailability of one tier component does not hinder the information flow from its tier t_p to the tier the inbound middleware belongs to. Like this, it maintains the connections and tries reconnecting until the leadership is ceased.

The inbound middleware has no specified state, but most implementations will likely contain some sort of channel for each tier $t_p \in T_p$ as well as state necessary to ensure proper reconnection.

For our implementation, we assume some kind of `ESTABLISHCONNECTION(tierId)` function which may be invoked when the inbound middleware is supposed (re)establish the connection to some preceding tier t_p with `tierId(t_p) = tierId`. We assume that until the implementation invokes `CLOSECONNECTION(t_p)` or the tier component crashes, the inbound middleware will automatically reconnect to t_p in case of a connection loss, iterating over t_p 's tier components. To close a connection in case the leadership ended, the inbound middleware additionally provides some kind of `CLOSECONNECTION(t_p)` function, which closes the connection to t_p . This already implies that the inbound middleware only allows at most one connection per preceding tier.

As the job of the inbound middleware is to process incoming messages, the maybe most important function it has to have is `RECEIVEMESSAGE(m)`, which receives messages via one of the inbound connections.

To send acknowledgments to the preceding tier, we also require a `SENDMESSAGE(m, t)` function, which sends a message m to tier t . Note that by the assumption of a connection-oriented network layer, we assume that either sending is retried automatically or the connection is closed, eventually, which then leads to a new connection.

All four functions, `ESTABLISHCONNECTION`, `CLOSECONNECTION`, `RECEIVEMESSAGE` and `SENDMESSAGE`, are often either provided by the operating system, the programming language's standard library or the networking framework. Otherwise, we require the inbound middleware of the concrete implementation to implement them appropriately.

4.4.6. The Outbound Middleware

As the inbound middleware of each tier actively establishes connections, the outbound middleware can act as some kind of server which waits for connections that are to be established

from other tiers $t_c \in T_c$ which are interested in messages produced by the tier t , the outbound middleware belongs to.

The outbound middleware has two important tasks. Firstly, it assigns a unique id to each message created by the state machine and, secondly, it has to make sure, that all tiers t_c that established connections to t , receive all unacknowledged messages. For optimal performance, this should happen in the order of the ids assigned to them.

If a consuming tier established a connection, we assume that a `ESTABLISHEDCONNECTION(conn)` function is invoked. `conn` has to contain at least the tier id of the consuming tier that established the connection. Furthermore, to send messages, we require a `SENDMESSAGE(m, t)` function which sends a message m to tier t . Finally, to receive (non-)acknowledgments, the outbound middleware also needs to provide a `RECEIVEMESSAGE(m)` function.

Note that we again assume that either resending of a message is done automatically by the network layer or the connection is closed, eventually.

4.4.7. Underlying File System

To store and recover from snapshots, we require a file system which consists of stable storage and should be accessible in three ways. First, implicitly, to retrieve a particular snapshot to recover from, second, to write a snapshot via a `WRITETOSTABLESTORAGE` function and, third, to delete old snapshots from the stable storage via a `DELETEFROMSTABLESTORAGE` function. All are assumed to be atomic and the file system is assumed to provide access to snapshots in an abstract way, resolving all data and address lookups transparently.

4.5. Operations

As now all building blocks of a tier component have been defined, this section will describe how the building blocks interact to actually provide a working implementation.

4.5.1. Bootstrapping

On startup, a tier component can be run from a well-defined initial state or from a snapshot which is stored on non-volatile storage. Startup from initial state is described in Algorithm 4.1.

Algorithm 4.1 Bootstrapping from initial state

Lines with a # are only necessary if the replicated log does not provide enforced order.

```

smState ← smInitState // Initial state machine state
lastLogId ← 0
nextMessageId ← 0
allMessages ← []
unAcknowledgedMessages ← {}
unhandledEntries ← []
#lastProcessedMessage ← {}
#unprocessedMessages ← {}
do // Set initial minSnap value
    committed ← COMMIT(minSnap:ti:0)
    while ¬(committed = ok ∨ committed = duplicate)
    mode ← normalOperation

```

4.5.2. Recovery

When a tier component recovers from a snapshot s , the procedure looks a bit different. It is assumed that the replicated log recovers automatically from some kind of snapshot and/or transaction log. The remaining recovery steps for the tier component are described in Algorithm 4.2.

To maintain consistency while the replicated log is pruned, a tier component t_i will recover from exactly the one snapshot s for which holds $s:\text{lastLogId} = \mathcal{L}(\text{minSnap}(t_i))$. Such a snapshot exists by algorithm 4.11.

As during the execution of the for-loop additional entries might be appended to *unhandledEntries*, it might take an indefinite amount of time until the loop is left. If this should become a serious problem, it is also possible to let the for-loop interfere with normal operation. Then, it only has to be ensured that the last log entry that existed at the beginning of the recovery is applied before switching to normal operation mode. However, if the recovery loop is executed clearly slower than new entries are appended to *unhandledEntries*, a real system will eventually run out of memory.

4.5.3. Normal Operation

Once the tier component is bootstrapped and/or has recovered from a snapshot, it resumes normal operation. There are four things that can happen concurrently and independently of each other:

1. a consuming tier wants to establish a new connection,

Algorithm 4.2 Recovery from a snapshot s

Lines with a # are only necessary if the replicated log does not provide enforced order.

```

smState ← s:state
lastLogId ← s:lastLogId
nextMessageId ← s:nextMessageId
allMessages ← []
unAcknowledgedMessages ← {}
# lastProcessedMessage ← s:lastProcessedMessage
# unprocessedMessages ← s:unprocessedMessages
mode ← recovery
unhandledEntries ← [ $m_i \in \mathcal{L} : i > \text{lastLogId}$ ] // In the order as they were committed to  $\mathcal{L}$ 
for  $m_i \in \text{unhandledEntries}$  do // Each time this loop could be left, it has to be evaluated
    // whether unhandledEntries is empty.
    unhandledEntries ← unhandledEntries −  $m_i$ 
    HANDLEENTRY( $i, m$ )
end for
mode ← normalOperation

```

2. the current tier is elected as leader,
3. the replicated log notifies the tier component that some component – which is currently elected as leader within the tier – successfully committed an entry to the replicated log which is now persistent and can be applied to the state machine or
4. a snapshot is taken and the log is pruned accordingly.

If none of these things happens, the variables of the tier component will not change.

Each of the four operations is now described in detail in the following subsections.

4.5.4. Connection Establishment

A connection between two tiers is established from the tier t which consumes the messages produced by another tier t_p . As soon as a tier component $t_i \in t$ is elected as leader, it has to establish a connection to *any* component within each t_p . The detailed procedure is given in Algorithm 4.4. The producer side then has to try to register the consuming tier by setting a auxiliary minACK entry and/or send all outstanding messages. To make sure the suffix is initiated correctly, the first message's init flag is set to true. This is described in Algorithm 4.3.

Algorithm 4.3 Connection Establishment at the Producer Side

```
function ESTABLISHEDCONNECTION(conn)
  if  $\nexists$ (minACK  $\leftarrow$   $\mathcal{L}$ (minACK(tierId(conn)))) then
    minACK  $\leftarrow$  SETINITIALMINACK(tierId(conn))
  end if
  while allMessages = {} do // Wait in case there is no message yet
  end while
  SENDMESSAGE(m  $\in$  allMessages : id(m) = minACK with init(m) = true, tierId(m))
  Tc  $\leftarrow$  Tc + tierId(conn)
  for m  $\in$  allMessages : id(m) > minACK do
    SENDMESSAGE(m, tierId(m))
  end for
end function

function SETINITIALMINACK(tierId)
  minMinACK  $\leftarrow$  GETMINMINACK() // This could also be set to
  // nextMessageId, probably leading
  // to fewer loop iterations in
  // average, but sacrificing the initial
  // registration property
  cond  $\leftarrow$   $\nexists$  $\mathcal{L}$ (minACK( $\cdot$ ))  $\vee$   $\nexists$  $\mathcal{L}$ (minACK(tierId))  $\wedge$   $\min$ ( $\mathcal{L}$ (minACK( $\cdot$ ))) = minMinACK
  while  $\neg$  COMMITIF(minACK:tierId:minMinACK, cond) do
    if  $\exists$ (minACK  $\leftarrow$   $\mathcal{L}$ (minACK(tierId))) then // Some other replica committed a mi-
    // nACK value for tierId
      return  $\mathcal{L}$ (minACK(tierId))
    end if
  end while // The minMinACK value read above was stale; try again
  return  $\mathcal{L}$ (minACK(tierId))
end function

function GETMINMINACK()
  if Tc =  $\emptyset$   $\vee$   $\nexists$  $\mathcal{L}$ (minACK( $\cdot$ )) then
    return 0
  else
    minMinACK  $\leftarrow$   $\infty$ 
    for tc  $\in$  Tc do
      minMinACK  $\leftarrow$   $\min$ (minMinACK,  $\mathcal{L}$ (minACK(tc)))
    end for
    return minMinACK
  end if
end function
```

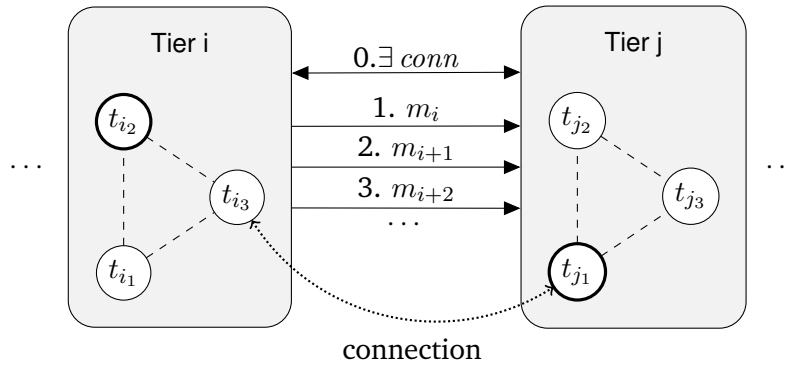


Figure 4.3.: A connection between tier i and tier j is established. Tier component t_{j1} is elected as leader (bold) within tier j and hence tries to establish a connection to some component within tier i, in this case t_{i3} . As soon as the connection is established, the respective tier component t_{i3} then starts sending from some message or resumes sending from the last message acknowledged by tier j.

Algorithm 4.4 Connection Establishment at the Consumer Side

```

function LEADERSHIPSTARTED()
  for  $t_p \in T_p$  do
    ESTABLISHCONNECTION( $t_p$ )
  end for
end function

```

4.5.5. Handling Incoming Messages

When the inbound middleware established some connection to a preceding tier, it is ready to handle incoming messages. This is specified in Algorithm 4.6.

We want to make sure that we do not commit an arbitrary message as first message for this preceding tier, which would then block all previously produced messages by the order enforced either by the replicated log or by the consensus adapter/state machine interface. Therefore, we have to make sure that the first message we commit is flagged as initial message by the

Algorithm 4.5 Connection Closing at the Consumer Side

```

function LEADERSHIPENDED()
  for  $t_p \in T_p$  do
    CLOSECONNECTION( $t_p$ )
  end for
end function

```

outbound middleware of the preceding tier. The fact that other messages – precisely the very first one sent after a successful reconnect – have the init flag set, too, will be ignored.

The very first message as well as all incoming messages are passed to the consensus adapter to commit the message to the replicated log. The consensus adapter might either confirm the commit, indicate that the message is already committed to the stable log or signal that the commit action has failed. In the first two cases, the inbound middleware will send an ACK message that acknowledges the reception and replication. If the message could not be committed to the replicated log, it will send a NACK message back to the sender.

Algorithm 4.6 Handling new messages received via one of the inbound connections

```

function RECEIVEMESSAGE( $m$ )
  if  $\mathcal{L}(\text{senderId}(m)) = \{\}$  then
    if  $\text{init}(m)$  then
       $\text{committed} \leftarrow \text{COMMIT}(m)$ 
    else
      SENDMESSAGE(NACK, senderId( $m$ ))
    return
  end if
  else
     $\text{committed} \leftarrow \text{COMMIT}(m)$ 
  end if
  if  $\text{committed} = \text{ok} \vee \text{committed} = \text{duplicate}$  then
    SENDMESSAGE(ACK, senderId( $m$ )) // With id(ACK) = id( $m$ )
  else
    SENDMESSAGE(NACK, senderId( $m$ )) // With id(NACK) = id( $m$ )
  end if
end function

```

4.5.6. Handling New Entries

Once the bootstrapping is complete and the log recovered, it is possible that the log replication mechanism will append new entries to \mathcal{L} . In this case, these new entries are handled as described in Algorithm 4.7. It does not matter whether the current tier component has committed the entries itself or they are proposed by another tier component. However, it is important that the entries are handled in the total order imposed by \mathcal{L} .

Besides changing its internal state, the state machine's specific code also has the ability to produce and send messages to all succeeding tiers by invoking the $\text{SEND}(\text{logId}, m)$ function (see Algorithm 4.8). To consistently prune the log (see below), it is necessary not only to pass the message that should be sent out, but also the entry id in \mathcal{L} that lead to the creation of the message.

Algorithm 4.7 Handling new entries in the replicated log \mathcal{L} .

Lines with a # are only necessary if the replicated log does not provide enforced order.

```

function NEWENTRY(logId, e)
  if unhandledEntries  $\neq$  [] then
    unhandledEntries  $\leftarrow$  unhandledEntries + (logId, e)
    // Enqueue the entry as recovery is not yet complete
  else
    HANDLEENTRY(logId, e)
  end if
end function

function HANDLEENTRY(logId, e) // To be executed by the state machine thread
# cond1  $\leftarrow$   $\exists$  lastProcessesMessage[tierId(e)]
#    $\wedge$  lastProcessesMessage[tierId(e)]  $\geq$  id(e)
# cond2  $\leftarrow$   $\exists$  lastProcessesMessage[tierId(e)]
#    $\wedge$  lastProcessesMessage[tierId(e)]  $\neq$  id(e) - 1
# if cond1 then // e has already been processed and delivered
#   return // Stop processing the entry
# end if
# if cond2 then // e must not be processed yet
#   unprocessedMessages[tierId(e)]  $\leftarrow$  unprocessedMessages[tierId(e)] + e
#   return // Stop processig the entry
# end if
  lastLogId  $\leftarrow$  logId
  APPLYENTRY(logId, e) // This invokes the state machine specific code
# lastProcessesMessage[tierId(e)]  $\leftarrow$  id(e)
# while  $\exists m \in$  unprocessedMessages[tierId(e)] :
#   id(m) - 1 = lastProcessesMessage[tierId(e)] do
#   // Check for further entries m that have been deferred to
#   // ensure id order and can be handled now
#   unprocessedMessages[tierId(e)]  $\leftarrow$  unprocessedMessages[tierId(e)] - m
#   APPLYENTRY(logId, m) // logId can be the one from above
#   lastProcessesMessage[tierId(e)]  $\leftarrow$  id(m)
# end while
end function

```

4.5.7. Managing Outgoing Messages

When the state machine produces a message, the outbound middleware has to ensure three things. Firstly, it has to send the message to all tiers that have established a connection to this tier component. Secondly, it has to give the message a unique, reproducible id that will be the same even when the tier component crashed and the message is sent during recovery. This is also the very same id that is assigned to this message content by all tier components within this tier. Finally, it has to manage the acknowledgments of the messages, which is important to be able to correctly prune the log. All this is described in algorithm 4.8.

A concrete implementation has to ensure that the outgoing messages obtain the correct ids. Therefore, it has to make sure that the send operations are processed in a thread-safe manner in the same order as they have been invoked by the state machine. One way this can be done is by having a dedicated single thread that processes them, which is what we assume in Algorithm 4.8.

4.5.8. Snapshots

Snapshots enable two things: fast recovery in case of a crash failure by replaying only the log entries that have been appended since the snapshot has been taken and pruning of the replicated log which otherwise would consume more and more memory as the system is running.

In our implementation, snapshots comprise not only of the state machine's state, but we also use them to capture the meta information necessary to ensure consistent enumeration of outgoing messages as well as enforced order in case the log does not provide this.

This leads to the following variables stored in a snapshot.

- **Log entry id**, *lastLogId*, the id of the last log entry applied to the state machine before taking this snapshot.
- **State machine state**, *smState*, is definable by the concrete implementation of the state machine.
- **State machine meta information.**
For each tier that this tier receives messages from:
 - *lastProcessedMessage*, the id of the last processed message.
 - *unprocessedMessages*, the messages that have been buffered until id order delivery is possible.
- **State of the outbound middleware.**

Algorithm 4.8 Managing outgoing messages that are created by the state machine

```

function SEND(logId, m)                                // m has to contain only the content to send
  id(m) ← nextMessageId
  senderId(m) ← ti
  nextMessageId ← nextMessageId + 1
  allMessages ← allMessages + (logId, m)
  for tc ∈ Tc do
    SENDMESSAGE(m, tc)                                // Pass m to the operating system network stack
    unAcknowledgedMessages[tc] ← unAcknowledgedMessages[tc] + id(m)
  end for
end function

function RECEIVEMESSAGE(m) // This is the version of the outbound middleware
  // For (N)ACKs: m contains the original message's id, the
  // tier id of the (non-)acknowledging tier and no content
  if content(m) = ACK then
    tc ← senderId(m)
    unAcknowledgedMessages[tc] ← unAcknowledgedMessages[tc] - id
    UPDATEMINACK(tc, min(unAcknowledgedMessages[tc]))
    // This is expensive and might be omitted unless
    // min(unAcknowledgedMessages(tc)) has changed significantly
    // since the last updateMinACK call
  else if content(m) = NACK then // Resend the message; possibly with init flag set
    if  $\mathcal{L}(\text{minACK}(\text{senderId}(m))) = \text{id}(m)$  then
      SENDMESSAGE(m' = allMessages[id(m)] with init(m') = true, senderId(m))
    else
      SENDMESSAGE(allMessages[id(m)], senderId(m))
    end if
  else
    // Discard m
  end if
end function

function UPDATEMINACK(t, minACK)
  COMMITIF(minACK:t:minACK,  $\mathcal{L}(\text{minACK}(t)) < \text{minACK}$ )
end function

```

- *nextMessageId*, the id of the next outgoing message to be produced by the state machine when recovering from this snapshot.

To ensure that snapshots are taken consistently, the state machine as well as the outbound middleware both have to be locked after fully processing a log entry, this means, after applying all outstanding messages that have been deferred to ensure id order and can be processed after the last applied entry and after delivering all outgoing messages to the middleware and assigning them with the corresponding ids.

We propose to simply use a single state machine thread and a single outbound middleware thread to do this. So let the state machine thread finish the complete processing of a new entry and enqueue the snapshotting job after it. Then the state machine thread forwards the unfinished snapshot to the outbound middleware thread which automatically leads to a correct ordering with respect to other outgoing messages. This behavior is assumed in Algorithm 4.9.

Algorithm 4.9 Taking a snapshot *s*

Lines with a # are only necessary if the replicated log does not provide enforced order.

```

function TAKESNAPSHOT( ) // To be executed only by the SM thread
    s:lastLogId ← lastLogId
    s:state ← STATE( ) // Defined by the concrete SM implementation
    # s:lastProcessedMessage[] ← lastProcessedMessage[]
    # s:unprocessedMessages[] ← unprocessedMessages[]
    s ← FINALIZESNAPSHOT(s) // This and the next line can be executed asynchronously
    return s
end function

function FINALIZESNAPSHOT(s) // To be executed only by the outbound middleware thread
    s:nextMessageId ← nextMessageId
    return s
end function

```

Snapshotting can happen either for each replica independently or for all replicas in a consistent manner. In case all replicas take consistent snapshots that are based on the same log id, the next proposal becomes redundant.

In case the replicas take snapshots independent from each other, we propose to use helper entries that are replicated using the log. These helper entries tell other replicas, until which log id the replicated log is not necessary anymore for the issuing replica to recover from a snapshot. We call them *minSnap*. As an example, a *minSnap*:3:42 entry would signalize all other replicas, that replica with id 3 only needs the log entries from 42 onwards to recover. This brings us to the interesting questions, which snapshot is necessary to recover from.

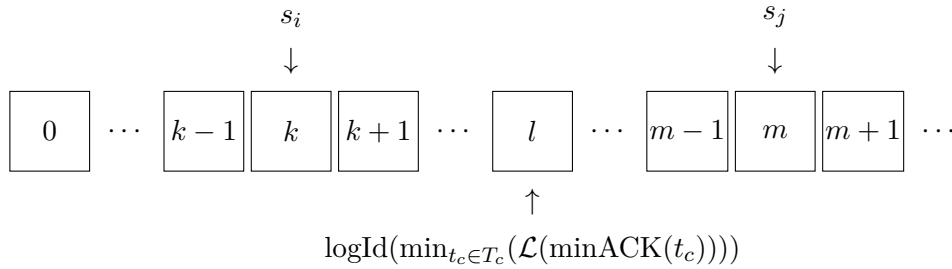


Figure 4.4.: Choosing which is the least recent snapshot that has to be kept.

Even though snapshot s_j is more recent than snapshot s_i , snapshot s_i has to be kept as only from it the first unacknowledged message $\min_{t_c \in T_c}(\mathcal{L}(\minACK(t_c)))$ is reproducible.

As the snapshot contains the current state of the state machine, the state machine per se can recover from any snapshot having the log entries following the one applied to the state machine before taking the snapshot. The meta information necessary to ensure id order of the messages applied to the state machine is contained in the snapshot, too. This also holds for the synchronously snapshotted next message id which is the first part of the vital information of the outbound middleware. The only critical point are the outgoing messages which are not yet acknowledged by the succeeding tiers and stored only in volatile memory managed by the outbound middleware. They need to be recreated by (re)applying the log entries to the state machine. So the snapshot to keep is the most recent one from which the first (with respect to the id assigned to the outgoing messages, which resembles the order in which they have been created) unacknowledged outgoing message, considering all currently known tiers, can be recreated. This is reflected in the \minACK log entries and illustrated in Figure 4.4.

The attentive reader will already have noticed the parallels between the messages buffered in the state machine to optionally implement enforced order and the outgoing messages created by the state machine which are stored by the outbound middleware until they are acknowledged by *all* succeeding tiers for this tier component. Both could be stored in the snapshot or recreated from the replicated log by restricting the log pruning procedure to make sure this volatile information can be recreated from some snapshot during recovery. We chose not to solve this in a uniform way as the state of the state machine with the meta information necessary for id order is an easy thing to capture, whereas the dynamic outbound connections make it hard to capture the outgoing messages in the same uniform way as the state and the state machine's meta information without restricting the possibly multi-threaded networking implementation too much. In general, however, this reflects two ways to solve the problem how to recreate necessary volatile information: one has to either save them in the snapshot and directly restore them to volatile memory or leave the original entries in the replicated log and recover them step by step.

Algorithm 4.10 Auxiliary functions for log pruning. They are only necessary if snapshots are taken independently for each tier component and thus denoted with a \diamond .

```

function GETMINMINSNAP()
   $\diamond$   $minMinSnap \leftarrow \infty$ 
  for  $t_k \in t$  do
     $\diamond$   $minMinSnap \leftarrow \min(minMinSnap, \mathcal{L}(\minSnap(\text{componentId}(t_k))))$ 
  end for
  return  $minMinSnap$ 
end function

function SETMINSNAP( $minSnap$ )
   $\diamond$   $\text{COMMIT}(\minSnap: \text{componentId}(t_i): minSnap)$  //  $t_i$  is the id of the tier component
end function

```

Algorithm 4.11 An example for log pruning which assumes independently taken snapshots. Lines with a \diamond/\bullet could either be removed (\diamond) or needed to be adapted (\bullet) for consistently taken snapshots.

```

function PRUNOLOG()
  if  $mode \neq \text{normalOperation}$  then
    return
  end if
   $s \leftarrow \text{TAKESNAPSHOT}()$  // Optional
   $\text{WRITESTABLESTORAGE}(s)$  // Optional
   $minMinACK \leftarrow \text{GETMINMINACK}()$ 
   $minMinACKEntryId \leftarrow \text{logId}(m \in allMessages : id(m) = minMinACK)$ 
   $s_{keep} : s_{keep}.lastLogId = \max_{s: s.lastLogId < minMinACKEntryId} (s : s.lastLogId)$ 
   $\diamond$   $\text{SETMINSNAP}(s_{keep}.lastLogId)$ 
   $\text{DELETEFROMSTABLESTORAGE}(s : s.lastLogId < s_{keep}.lastLogId)$ 
   $\diamond$   $minMinSnap \leftarrow \text{GETMINMINSNAP}()$ 
   $\bullet$   $\text{PRUNOLOG}(\{m : \text{logId}(m) < minMinSnap\})$ 
   $\text{PRUNEAUXILIARYLOG}()$ 
   $allMessages \leftarrow \{(\cdot, m) \in allMessages : id(m) > minMinACK\}$ 
   $unAcknowledgedMessages(\cdot) \leftarrow \{id \in unAcknowledgedMessages(\cdot) : id > minMinACK\}$ 
end function

```

Another interesting thing is, that when the smallest minACK value is determined, this can be done at any moment without any precautions by just evaluating the currently committed values of the replicated log. The same holds for the determination of the optional minSnap values for log pruning (see below). This is due to the fact that the value for each entry can only increase. Furthermore, minACK entries for new tiers are only considered valid and have an effect outside the tier if they increase the minimum of all minACK entries. This means that the minimum of all *valid* entries can only increase, too. In total, this yields monotonicity for the minimum of all minACK/minSnap values. This monotonicity makes sure that a view only leads to less pruning or fewer deleted snapshots, but working with it never harms consistency.

4.5.9. Compacting the Replicated Log

Firstly, it does not matter how or when log compaction is triggered. This could be done on a periodic basis or from a central administration server that sends requests that initiate the compaction of the log.

When the log is compacted or pruned, old entries are deleted from the log. This can happen for each replica independently or in a replicated manner. It only has to be taken care that each of the replicas can recover from a snapshot to first necessary state. This is either known as all snapshots are taken in a consistent way for all replicas or it can be determined by the minimum of the tier components' minSnap entries.

Note, that, in the context of pruning, with entries only the actual message entries are meant, but not the auxiliary minACK and minSnap entries. As we only access the very latest one of them, we only have to keep the minACK/minSnap entry with the largest log id for each tier or tier component respectively. This pruning of auxiliary entries is abstractly denoted by PRUNE_AUXILIARY_LOG.

The pruning of the actual message entries is invoked via the function PRUNE_LOG. In addition to the message range passed to the function, it has to make sure, that for each producing tier, at least one message entry is kept to not require a further initial message according to Algorithm 4.6.

4.5.10. Removing Old Snapshots

Snapshots that are not needed for recovery anymore – as determined by the way described above – can be deleted. This can happen in the background and for each tier component independently.

5. Concrete Implementation Using ZooKeeper

This chapter elucidates how we came from the abstract implementation, we described in the previous chapter, to actual Java code. In particular, we discuss the choice for the replicated log and possible alternatives.

5.1. In Search of a Replicated Log

The actual implementation of the protocol makes use of ZooKeeper as a service that provides a replicated log as well as leader election. This section shows possible alternatives to ZooKeeper and explains why ZooKeeper was chosen eventually.

As explained in chapter 2, there are two general possibilities to obtain a replicated log: active replication protocols such as consensus protocols like Paxos or Raft and passive or primary-backup replication protocols like ZooKeeper's Zab and Viewstamped Replication. While it would be possible to implement the abstract implementation using a passive replication scheme, too, an active replication scheme or its equivalent, we eventually implemented on top of ZooKeeper, much better fits our needs as we actually want replicated access to the state machine commands and not only to the state updates.

5.1.1. Paxos

While Paxos is by far the oldest and most general active replication consensus protocol, its generality also lead to famous problems in grasping all corner cases and in consequence various error-prone implementations [CGR07]. Furthermore, as the replicated log is really the heart of the tiered state machine replication protocol, the replicated log implementation was supposed to be as stable and as proven as possible. While there are a few Paxos implementations and libraries in Java, none of them is used widely enough to ensure this.

The probably most famous use of Paxos in the industry is as part of Chubby [Bur06] and Spanner [Cor+13], the coordination service and distributed, replicated database Google's services are built upon. However, while there are papers documenting Chubby's and Spanner's implementation details, the source code is closed source.

5.1.2. Viewstamped Replication

Being developed at about the same time as Paxos, Viewstamped Replication is the first well-known passive replication consensus protocol. It heavily influenced the design of both, Zab and Raft, but had some practical flaws [RSS15], which might explain why it is hard to find more than one single Java implementation of it. As with Paxos, the only implementation we were able to retrieve is apparently not used in any industry product and in consequence was seen as not proven enough.

5.1.3. Raft

Raft is the most recently published well-known consensus algorithm. It was designed with understandability as a primary goal. This leads to an allegedly easy implementation, the availability of implementations in most common programming languages [Ong08] and heavy industry use such as in etcd, a “distributed, consistent key-value store” [Cor08] by CoreOS.

Unfortunately, most Java implementations are not more than experimental which might be explained by the use of Raft implementations as an exercise in distributed systems courses. In total, only three raft implementations seemed to be ready for actual use as a library or as a foundation to build a custom protocol upon, which were Allen George’s libraft, Jordan Halterman’s copycat and jgroups-raft by Bela Ban.

Libraft

Libraft [Geo14] seemed to be simple at the first glance, but was not running stable and does not seem to be actively supported anymore. This is why libraft was discarded as a candidate to for the replicated log.

Copycat

The next candidate we considered is copycat [HH08], which is supposed to be a full-featured general purpose implementation of Raft in Java. Copycat looked very promising and an early version already ran stable for long periods and even under load. However, while this early version utilized a replicated log as part of the Raft consensus algorithm, there was no way to access it safely without hurting the consistency promises provided by Raft. Noticing ongoing changes in the public source code repository, we contacted copycat’s main author and maintainer, Jordan Halterman. Halterman admitted the unavailability of the log, but sketched a state-machine based implementation of a replicated log for the upcoming stable release which was planned to be released on June 21, 2015. So, the log was still not supposed to be directly accessible, but the state-machine based implementation seemed to be simple enough

to provide a fast and safely accessible replicated log anyway. Building such a state-machine based implementation based on the source code available at the time in various branches, we found out that the code had become fairly unstable. As time passed by and we further exchanged some emails with Halterman, it became clear, that even on June 21, a stable release would be far in the future. While we still think that copycat is probably the best candidate to become a general purpose Raft implementation which could be used to provide a replicated log, we thus decided not to use it for our work. As of August 31, 2015, copycat finally has been released in version 1.0.0.

Jgroups-Raft

The other Raft implementation we evaluated in depth is Jgroups-Raft [BKL20] which is maintained by Bela Ban. Jgroups-raft is based on the group communication framework Jgroups [JG] of which Bela Ban is the maintainer, too. It, too, looked very promising in our first tests, but using it under load showed instabilities – for instance as the leader timed out. Examining the source code, we saw that Jgroups-Raft deviates from the proposed Raft implementation in the original paper in various points, most significantly using Jgroups' leader election. Separating leader election from the rest of the Raft implementation, but using the same network connection lead to election timeouts as the election protocol's keep-alive packets were delayed arbitrarily under load. As this was not the only problem and the source code deviated from the Raft protocol in other points, too, we decided not to use Jgroups-Raft either.

So, while we still think, that Raft is a great protocol for building a replicated log, the unavailability of proven Java implementations at the time of implementation lead us to consider what is probably the industry's Java standard tool for solving consensus and similar problems: ZooKeeper.

5.1.4. ZooKeeper

ZooKeeper [ZK] is a general purpose coordination service which offers a tree-like node structure which does not directly represent a log. However, ZooKeeper has been used by several big companies which rose hope that the current version is well-tested. Furthermore, it offers great consistency guarantees, compensating the complicated internal data structure, and enables auxiliary information be directly stored and then easily retrieved by all replicas. While, at first, ZooKeeper was only thought of as a fallback option, it finally became our first choice, working stable even under great load and over long periods of time.

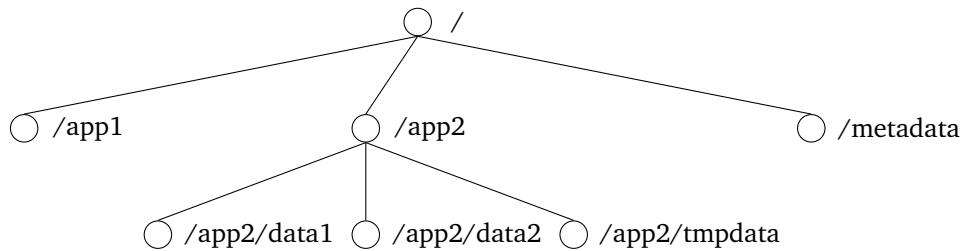


Figure 5.1.: A general example how ZooKeeper’s hierarchical data structure is intended to be used by multiple distributed applications as a shared and fault-tolerant coordination service.

5.2. Employing ZooKeeper as Replicated Log

Each tier component includes its own instance of a ZooKeeper server (ZKS) which acts as a replica of ZooKeeper’s data structure. The ZooKeeper source code only has been modified minimally to obtain better performance. The consensus adapter communicates with the ZKS in two ways: it has a regular client connection which provides ZooKeeper’s consistency guarantees, but also gets notified whenever a new transaction changes the internal log of the ZKS.

ZooKeeper maintains a tree-shaped internal data structure, which is designed to provide developers which a structure they already know from file systems. While small amounts of data can be stored by naming and structuring nodes appropriately, the actual way to store larger amounts of data is within a node.

While the tree-shaped data structure might be great for application developers, the feature most important to use is that all changes in the data structure are replicated using Zab (see chapter 2) before they become effective. This yields a replicated log that is additionally logically structured as a tree or vice versa.

5.2.1. Committing New Log Entries

New entries are committed via the client connection which provides FIFO order and very comfortable error semantics.

As the client connection guarantees FIFO ordering, this could offer an easy way to implement the enforced order property for committed log entries. One only would have to make sure that only a single thread that receives and commits entries. However, as committing one entry could fail, but committing the very next one could succeed, this would imply to use synchronous client calls. Given the need for replication, a synchronous commit takes about 100ms, thus

limiting the throughput to about 10 messages per second. Fortunately, ZooKeeper offers two alternatives.

Firstly, it supports synchronous, but transactional commits. These could be used in combination with multiple parallel connections to ensure FIFO order, too. This is as simple as specifying an existence check for the previous message and only commit the new entry if this is passed.

The other possibility, we eventually chose, is to use asynchronous commits and give up enforced order. In a few tests, giving up enforced order for the log and extending the state machine to cope with out-of-order messages seemed not only easier to implement, but also performed clearly better.

5.2.2. Storing Millions of Long Message Ids in a Tree Structure

As ZooKeeper's log is not directly exposed to the ZooKeeper client, we had to use ZooKeeper's tree structure and map the message ids to it. Messages are therefore stored as nodes in ZooKeeper with `/log/` serving as root node. To distinguish messages by tier, they are then put as children of a node having their sender as name, i.e. `/log/senderTierId/`.

The first approach was then to create a child node having the message id as node name, i.e. `/log/senderTierId/messageId`. Testing the implementation over longer runtimes showed that performance degrades badly when the number of stored messages increases and we finally found, that it is a known issue, that ZooKeeper cannot handle more than thousands of child nodes and even fails when listing too many child nodes [Hsi04]. This led to the hierarchical structure described in the following paragraph.

In the current implementation, message ids are mapped to a hierarchical structure by splitting up the number to thousands, millions, billions etc. It is then arbitrarily defined that the maximum message id is 999,999,999,999,999 which is enough to process 1 million messages per second for more than 30 years. This leads to an additional tree structure – besides the log root node and the sender's tier id – with a depth of 5. Each level then corresponds to three digits in the message id. This means that message m_0 is stored at node `/log/senderTierId/000/000/000/000/000`, m_{123} is stored at node `/log/senderTierId/000/000/000/000/123`, m_{1000} is stored at node `/log/senderTierId/000/000/000/001/000` and so on.

5.2.3. Listening for New Entries

The official way how clients are supposed to read data from ZooKeeper is by reading the data stored at a node or listing its children. Simple invocations can be enhanced by watches, which invoke a callback function whenever data changes, a child is added/deleted or similar changes

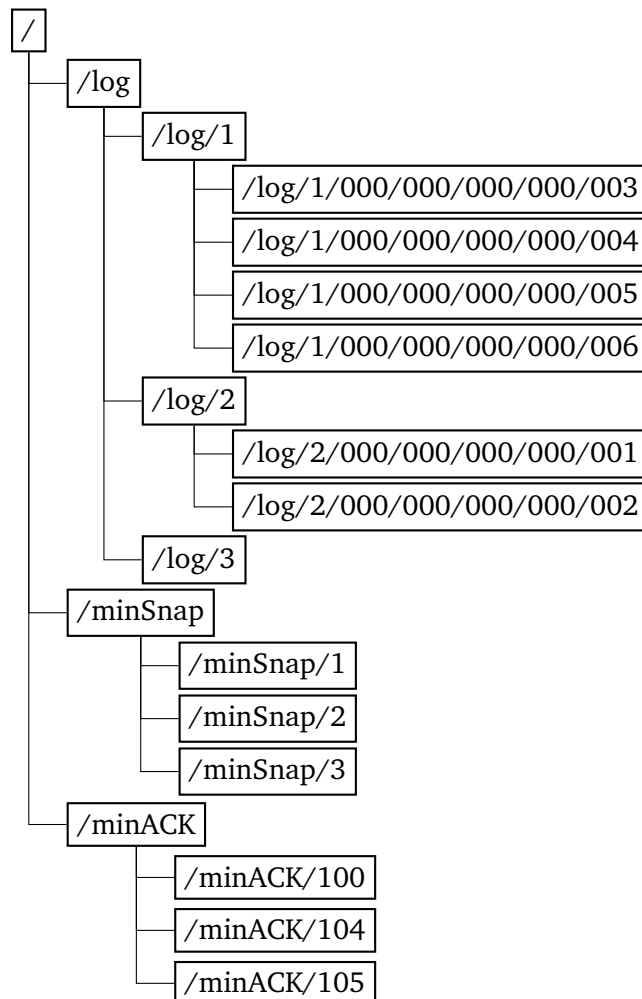


Figure 5.2.: ZooKeeper’s internal tree-shaped data structure utilized as a replicated log.

The tree data structure enables storing auxiliary entries and the actual log in separate logical subtrees. Then, for each tier there is a subtree representing a dedicated log.

While the whole data structure is accessible like a file system tree, it is in fact additionally totally ordered by a replicated log to which all changes in the tree have to be committed to become effective.

occur at the data node that this watch has been issued for. Unfortunately, watches only lead to a notification *once*.

For our implementation, this means, to get all changes of the log, we have to query some node for its children, issuing a watch at the same time. As soon as this watch invokes the callback, we have to query the node for its children again, yielding one or more changes, and set a new watch on the same, but also on sibling nodes. Then, we additionally have to read the child nodes to get the data the message contained. We tried this, but besides being really tedious, it also was truly slow.

Fortunately, there is a central point in the `ZooKeeperServer` class, which is used by both, leaders and followers of Zab, each transaction that changes the internal data structure after being committed to the transaction log has to pass through. By adding the notion of a change listener, we are thus able to get a copy of each transaction containing the log id, the path of the node that is affected and the content of the node in case the transaction creates a node or changes its data. This data can then easily be consumed and filtered by the consensus adapter without measurably hurting the performance. In particular, we filter only for create transaction that fit the naming scheme we introduced above.

5.2.4. Setting minACK and minSnap Values

Setting and updating minACK and minSnap values requires the conditional `COMMITIF` function. This can easily be realized in ZooKeeper using two features. The first is transactional commits with a check for existence. The second is that ZooKeeper supports version-dependent writes. This means besides getting data, each node also has a version that is monotonically increasing. Each write can then not only contain content, but also specify the version it is expecting. In case this version changed, the commit will be denied and the client application – which is the consensus adapter in our case – can handle this failure gracefully.

minACK nodes are then stored for each consuming tier at `/minACK/consumingTierId`. minSnap nodes are stored for each tier component at `/minSnap/tierComponentId`.

5.2.5. Log Cleaning in ZooKeeper

While ZooKeeper does its own snapshotting and log cleaning, it is not aware of the application-specific snapshots we take in parallel. Thus, while committed messages will be removed from the log, they will still exist in the data tree as nodes. Thus, we need to manually remove these nodes from ZooKeepers internal data structure as soon as we do not need them anymore. While this will issue additional log entries in ZooKeepers internal log, they will automatically be removed eventually by ZooKeeper thus removing all traces of a message that once has been committed to the replicated log.

On the other hand, an advantage of ZooKeeper's data structure is, that we do not need to implement any auxiliary log pruning as ZooKeeper automatically only keeps the most recent value for each auxiliary entry. This is due to the fact that each auxiliary entry corresponds to a exactly one data node.

5.2.6. Removing ZooKeeper's Snapshots

A final minor issue with ZooKeeper is that it does not automatically remove old snapshots by default. Instead, this is supposed to be done either via specifying the `autopurge` option or by manually by the administrator of a ZooKeeper cluster. For this, we simply refer to ZooKeeper's administrator guide [ZKAG].

5.2.7. Leader Election

As we already had ZooKeeper server instance for each tier component, it was straight forward to also use ZooKeeper to elect a leader. While it would have been possible to do this using ZooKeeper as a coordination service, we decided to slightly modify the ZooKeeper server again to access the primary status of the server instance. We then adopted the primary as a leader among the replicas.

6. Proof

In this chapter, we will prove the correctness of the abstract implementation described in chapter 4 with respect to the properties stated in chapter 3. Therefore, we first introduce some prerequisites. Then, we try to break down the whole proof into several smaller lemmata, that are proven step by step. Finally, we can derive the proofs for the actual protocol properties directly from the lemmata proven before.

6.1. Preliminaries

6.1.1. Replicated Log

The replicated log is assumed to provide strict and complete total order. This ensures that if some replica sees $\text{logId}(m_i) < \text{logId}(m_j)$, all replicas will order m_i before m_j . Furthermore, we even require that $\text{logId}_k(m_i) = \text{logId}_l(m_i)$ for all messages m_i and any two replica k, l . This last property, however, is already given for all well-known consensus protocols.

6.1.2. State Machine

As already stated above, the state machine is assumed to be deterministic and may only produce a finite number of messages for each entry that is applied.

6.1.3. Total Order of Incoming and Outgoing Messages

At several points, we will require that incoming as well as outgoing messages are strictly totally ordered. This strict total order has to be the same for all tier components of a tier and also has to be recreated when a machine recovered from a crash failure. Let us now explain how this order is achieved and how it propagates across a tier.

The total order of the incoming messages is imposed by committing them to the replicated log and passing them to the state machine in the order that is determined by the log. This leads to the following guarantee for all incoming messages im_i and im_j

$$im_i <_i im_j \vee im_j <_i im_i$$

with $<_i$ determining the total order of the replicated log. This also gives us the totally ordered set of incoming messages IM .

All outgoing messages are created by the state machine, which is deterministic and produces only a finite number of messages for each input. Each of the produced messages may be ordered as, by the deterministic nature of the state machine, they will be produced in a deterministic order. For outgoing messages $om_{i_0}, om_{i_1}, \dots, om_{i_{n-1}}$ that are created by applying incoming message im_i to the state machine, it thus holds

$$im_i \rightsquigarrow om_{i_0} <_{o_i} om_{i_1} <_{o_i} \dots <_{o_i} om_{i_{n-1}}$$

with $<_{o_i}$ determining the order of outgoing messages for incoming message im_i . We can then also see the set of outgoing messages for each incoming message im_i as a strictly totally ordered set OM_i .

Using the totally ordered set of incoming messages, IM , as an index set for the set of totally ordered sets of outgoing messages $OM = \{OM_i : im_i \in IM\}$, we obtain the so-called “natural” strict total order

$$om_i <_o om_j \Leftrightarrow (om_i, om_j \in OM_k \wedge om_i <_{o_k} om_j) \\ \vee (om_i \in OM_m, om_j \in OM_n \wedge im_m <_i im_n)$$

for all outgoing messages om_i and om_j . If we use the notation $<$ for messages, this actually stands for this natural strict total order $<_o$.

6.1.4. Unique and Reproducible Message Ids

For consistency, it is essential, that, for each tier, each message has a unique id, no matter from which tier component it is received. Furthermore, the unique id must be the same even when one or more machines recovered from a crash failure.

The uniqueness of the id of outgoing messages can be trivially derived exploiting their natural total order $<_o$:

$$id(om_i) := \sum_{\substack{om_j \in OM: \\ om_j <_o om_i}} 1 = |\{om_j \in OM : om_j <_o om_i\}|$$

While the replicated log ensures that this natural order in fact exists by imposing an order for incoming messages and all replicas initiate their state machine with an identical state, this does not trivially hold for recovery.

For recovery from a snapshot, however, we can see the recovering node as just another replica that applied all log entries up to the snapshot at once by restoring the state machine’s state from the snapshot. If it then continues to apply all entries that are not included in the snapshot, it actually becomes the same as a slow replica that is lagging behind. By this argument, the message ids will also be the same during recovery, given that recovery is properly implemented.

6.1.5. Further assumptions

To not unnecessarily lengthen the following proof to deal with all alternatives and eventualities, we will assume the following:

1. Snapshots are taken by each tier component independently and, thus, tier components use minSnap entries to determine which part of the log can be pruned.
2. The replicated log does *not* provide enforced order, but this is done directly before a new entry is applied to the state machine (as it is described in Algorithm 4.7).

6.2. Lemmata

This section contains the lemmata necessary to later prove the actual properties that are required by the problem statement.

Lemma 6.1 *At each tier, it holds: For each consuming tier t_c , the minACK value (stored in the log \mathcal{L}) is monotonically increasing.* □

PROOF (LEMMA 6.1) The replicated log is assumed to be persistent, that is, its values entries survive crash failures. So the only possibility to change the minACK value for a consuming tier t_c is by invoking the UPDATEMINACK() method, which tries to append a new – more recent – entry with a new value. This method, however, only commits the new value with the condition that it increases the existing value. ■

Lemma 6.2 *At each tier, the minimum of all minACK values is monotonically increasing.* □

PROOF (LEMMA 6.2) The minimum of a set may change in three ways: Adding a member that is smaller than the current minimum or removing or changing the member that constituted the current minimum.

As we only allow to add tiers and log pruning always preserves the most recent entry for each tier, the second possibility can be discarded for the set of minACK values which is persistently stored in the replicated log. The last possibility, changing the current minimum member, only increases the minimum by Lemma 6.1. What remains to verify is that adding new minACK values only may increase the value, too. As this only happens via SETINITIALMINACK in Algorithm 4.3, which issues a conditional commit specifically tailored to ensure this monotonicity, it is impossible to add a value smaller than the current minimum. This proves Lemma 6.2. ■

Lemma 6.3 *For each tier component $t_i \in t$, it holds*

$$\mathcal{L}(\text{minSnap}(t_i)) < \text{logId}(m : \text{id}(m)) = \min_{t_c \in T_c} (\mathcal{L}(\text{minACK}(t_c)))$$

at any point in time. □

This invariant states that the `minSnap` value of each component $t_i \in t$ will always be less than the log id of the first message m that is produced at tier t and is not yet acknowledged by some consuming tier. Looking back at Algorithm 4.8, this log id is the log id of the incoming message whose application to the state machine lead to the production of m .

PROOF (LEMMA 6.3) `minSnap` values are only changed when the log is pruned (Algorithm 4.11) and each tier components changes only its very own value. Then, there is the first snapshot to keep, s_{keep} , which is constructed such that its `lastLogId` is smaller than the log id of the message with the id which is minimum of all `minACK` values. Furthermore, it is s_{keep} 's `lastLogId` which becomes the new `minSnap` value of this component. This shows that the implementation only sets `minSnap` values that are – at the time they are set – smaller than the smallest `minACK` value.

We may ignore the case after initial startup as then the `minSnap` value is 0 by Algorithm 4.1 and the log ids are assumed to be non-negative which inhibits log pruning with the condition given in Algorithm 4.11.

According to Lemma 6.2, the minimum of all `minACK` values may only increase. Combining both, `minSnap` values that are only set to be smaller than the minimum of all `minACK` values and a monotonically increasing minimum of `minACK` values, the invariant stated by Lemma 6.3 becomes evident. ■

Lemma 6.4 1. Log entries e , which are needed for recovery,

$$e : \text{logId}(e) > \min_{t_i \in t} (\mathcal{L}(\text{minSnap}(t_i)))$$

will never be removed from the log.

2. For each tier, the `minSnap` value is monotonically increasing.

3. For each tier, the value of `lastLogId` is monotonically increasing during normal operation. □

As these three properties are interdependent, we have to treat and show this as one lemma.

PROOF (LEMMA 6.4) Firstly, assume the tier component starts from the initial state. Then, `lastLogId` is only changed by Algorithm 4.7, whose `NEWENTRY` function is only called in the order of the log entries. Those log ids are thus monotonically increasing. Furthermore, as long as the tier component does not crash, it may take snapshots and prune the log. As the snapshots are taken, their `lastLogId` field may only increase over time. Then, the `minSnap` value is changed, but as we take the maximum of all snapshots that are bounded by the entry id of the message with the id that is the same as the monotonically increasing minimal `minACK` value and only remove snapshots with a smaller `lastLogId`, the `minSnap` value only increases, too. Finally, we only prune the log according to Algorithm 4.11, which ensures that we only prune values smaller than the `minSnap` value. As the `minSnap` value is monotonically

increasing, the `minSnap` value is always a lower bound for the part of the log that is not yet pruned. This proves all three properties for the case that the tier component does *not* crash.

After the first crash, we then can use the above properties to recover from the snapshot whose `lastLogId` field has the value of the current `minSnap` value. As the set of entries whose `id` is larger than the `lastLogId` only may have grown during the crash, reboot and recovery (remember that we were not allowed to prune any of these values), the recovery will for sure increase `lastLogId` to a value higher or equal to the last `lastLogId` value before the crash. Then, `mode` will be set to normal operation, again, and this directly yields the monotonicity of `lastLogId` for the first recovery. As Algorithm 4.11 only changes anything if the tier component is in normal operation mode, we can now use the monotonicity of `lastLogId` to repeat the argument from above to show that, indeed, all more recent snapshots will have monotonically increasing `lastLogId` fields, the `minSnap` value may only increase over time as this is the maximum of existing snapshots' `lastLogIds` and, finally, no log entry needed for recovery is removed.

As we just proved all properties for the recovery from some state where they were true, we can simply repeat the argument to prove by induction that they will hold at all time. ■

Lemma 6.5 *For each tier component, the message m with the smallest `minACK` value,*

$$m : \text{id}(m) = \min_{t_c \in T_c} (\mathcal{L}(\text{minACK}(t_c)))$$

will be reproduced during recovery. □

PROOF (LEMMA 6.5) As each tier component $t_i \in t$ recovers from some snapshot s whose `lastLogId` field has the value of the current `minSnap` value for this tier component, we can simply put together Lemmata 6.3 and 6.4. By Lemma 6.3, $s:\text{lastLogId}$ will always be smaller than the log `id` of the entry $e_{\text{minMinACK}}$ that caused the state machine to produce the message with the smallest `minACK` value. Lemma 6.4 then ensures by $\min_{t_j \in t} (\mathcal{L}(\text{minSnap}(t_j))) \leq \mathcal{L}(\text{minSnap}(t_i)) = s:\text{lastLogId}$ that $e_{\text{minMinACK}}$ as well as all entries $e : s:\text{lastLogId} < \text{logId}(e) < \text{logId}(e_{\text{minMinACK}})$ are handed over to the state machine during recovery according to Algorithm 4.2 as they are for sure not pruned. This, however, will make the state machine reproduce the message with the smallest `minACK` value as the state machine produced this message before the crash by applying the same sequence of messages. Furthermore, this renders the checks for enforced order redundant as the entry sequence must have passed them before the crash to produce m . ■

Lemma 6.6 *At each tier component that is functionally up, the message with the smallest `minACK` value (see Lemma 6.5) will be contained in `allMessages`.* □

PROOF (LEMMA 6.6) There are two possibilities how `allMessages` is filled: either during normal operation or during recovery.

For the normal operation case, all produced messages are stored in *allMessages* according to Algorithm 4.8. So, there, we have to make sure, that the message with the minimal minACK value is not removed from *allMessages* during log pruning. As this is only done in Algorithm 4.11 with the minimal minACK value as a upper bound, Lemma 6.6 directly follows from the monotonicity the minimal minACK value, which may only increase.

During recovery, Lemma 6.5 holds and there is no pruning of *allMessages*, so when recovery has completed – which is the definition of a tier component that is functionally up – Lemma 6.6 will hold, too. ■

Lemma 6.7 *If for some consuming tier t_c the respective minACK value $\mathcal{L}(\text{minACK}(t_c))$ at tier t is i and at least one tier component t_j at tier t as well as at least one tier component $t_{c,k}$ at tier t_c is functionally up sufficiently long, message m_i will eventually be committed to the replicated log of t_c .* □

PROOF (LEMMA 6.7) By the properties of the leader election algorithm, which eventually will elect one of the functionally up tier components in t_c , let $t_{c,k}$ be the tier component in t_c which is elected as leader within t_c . Furthermore, t_j will be the tier component in t to which $t_{c,k}$ will eventually succeed to establish a connection. This is guaranteed by the channel properties of the network and the iterative reconnections attempted by the inbound middleware

Also, by the properties of the channel, the connection will eventually be open for a sufficiently long period of time to allow the following to happen (according to Algorithms 4.4 and 4.3):

- (0. $t_{c,k}$ and t_j establish the connection.)
- 1. t_j starts sending from m_i which is in *allMessages* according to Lemma 6.6.
- 2. The channel will deliver m_i at $t_{c,k}$.
- 3. $t_{c,k}$ successfully commits m_i to the replicated log of t_c .

The last point is not obvious. However, if m_i is the first message sent via a connection, it will have the init flag set to true. This holds for the first sending according to Algorithm 4.3 as well as for resending according to Algorithm 4.8. Thus, it will eventually be committed to the log.

If m_i is not the first message sent via the connection, but its id is the one of the minACK value at t , this implies that there has been another message successfully committed to the replicated log of t_c before – otherwise the minACK value would be lower – and hence the log for t cannot be empty. In this case, m_i will eventually be committed to the log, too. ■

Note that the proof only considers the “nice” case in which both tier components as well as the channel will behave such that connection establishment, delivery and commit operation of m_i succeed. While we do not specify when this happens, we know by the node and network properties required for progress that such a “nice” case will be reached eventually. Meanwhile,

there could be plenty of failure cases which are all covered either by the recovery of the tier components or by the reconnect property of the implementation of the inbound middleware in combination with the best-effort implementation (see [GOS98]) of the connection-based channel.

Lemma 6.8 *With the same properties as for Lemma 6.7, message m_i for the minACK value i will eventually be acknowledged by tier t_c . The minACK value then is increased by at least one.* \square

PROOF (LEMMA 6.8) In the first case, tier t receives an acknowledgment for message m_i . Then, i is removed from $unAcknowledgedMessages(t_c)$ and t_c 's minACK value is set to the new minimum of $unAcknowledgedMessages(t_c)$. By the strict total order of messages, however, this increases the minACK value by at least one.

In the other case, tier t will resend m_i according to Lemma 6.7 until it receives an acknowledgment from t_c and succeeds to increase the respective minACK value. By the recovery, reconnect and channel properties – the argument is the same as for m_i 's delivery in the proof of Lemma 6.7 – this acknowledgment will eventually be successfully received and processed at tier t , increasing the minACK value by at least one (see argument for the first case). \blacksquare

Lemma 6.9 *If some tier t produces some message m_i , m_i will eventually be committed to the respective replicated log by each consuming tier t_c whose minACK value is less or equal than i .* \square

PROOF (LEMMA 6.9) There are two things that could happen with m_i for each consuming tier t_c . If t_c eventually receives and commits m_i , this is trivial.

If that does not happen, the minACK value will eventually be i . This is due to Lemma 6.1, Lemma 6.7, Lemma 6.8 and the requirements for progress we stated in the system model. Then, however, there is – by the system model – at least one tier component within each t_c that is eventually functionally up so this directly follows from Lemma 6.7. \blacksquare

Lemma 6.10 *The first message m from a tier t that is committed to the log of some consuming tier t_c has a log id less or equal to the minACK value of t_c at t .*

$$\text{id}(m = \arg \min_{\substack{m \in \mathcal{L}^{t_c}: \\ \text{senderId}(m)=t}}(\text{logId}(m))) \leq \mathcal{L}^t(\text{minACK}(t_c))$$

This ensures that due to message reordering or failed commits no later message is committed first and then, to enforce order, all previous message are prevented from being processed as this would hinder both, proper log compaction and the initial registration property.

PROOF (LEMMA 6.10) According to Algorithm 4.6, only a message with an init flag set to true may become the very first message committed to the log. However, at each point where m is (re)sent, its init flag is only set to true if its id is the one of the current corresponding minACK value. That is the case in Algorithm 4.3 when m is the very first message sent, as the minACK value has just been defined then, or in Algorithm 4.8 when m is resent.

Having increasing monotonicity of minACK values in mind (Lemma 6.1), the above inequality becomes obvious. ■

Lemma 6.11 *Let $m \in \mathcal{L}$ the first entry for some preceding tier t_p that is successfully committed to the replicated log of tier t . Then, m will eventually be delivered to all tier components $t_i \in t$ that are eventually functionally up.* □

PROOF (LEMMA 6.11) If t_i does not crash, it directly follows from the properties of the replicated log or the respective consensus adapter that method `NEWENTRY` in Algorithm 4.7 is invoked. Then, there are two possibilities depending on whether `unhandledEntries` is empty or not.

1. `unhandledEntries` is not empty. Then m is stored in `unhandledEntries`, too. This happens atomically with the check whether `unhandledEntries` is empty. The only way that `unhandledEntries` is filled is during recovery (algorithm 4.2). If, however, `unhandledEntries` is non-empty, recovery has not yet completed and the recovery loop will eventually pass m to `HANDLEENTRY` (we have only a finite log and `unhandledEntries` is an ordered set that is processed in the order of the log entries' log ids). This leads to 2.
2. `unhandledEntries` is empty or `HANDLEENTRY` is invoked during recovery. Then, recall that m is the first message committed to the log for t_p . This implies that there is no `lastProcessedMessage[tierId(t_p)]` as it is only set in Algorithm 4.7, whose `NEWENTRY` function is invoked with entries ordered by their log id, or during recovery. During recovery, `lastProcessedMessage[tierId(t_p)]` is neither set as this would imply – with the same argument as for normal operation – that there was some entry before m , which objects the assumption. Thus, m will be directly applied to the state machine which equals delivery.

If t_i crashes before it delivered m , its `lastLogId` was for sure less than `logId(m)` by Algorithm 4.7. Then, there will be some snapshot s from which t_i eventually succeeds to recover as it is assumed to be eventually functionally up. As snapshots are only taken during normal operation, it holds `logId(m) > s:lastLogId`. Then, however, m will be delivered during the recovery according to Algorithm 4.2 – in which `unhandledEntries` is chosen just like that – and due to point 2. from the argument we just developed for the non-crashing case. ■

Lemma 6.12 *Let $m \in \mathcal{L}$ be some log entry that is successfully committed to the replicated log of tier t . Then, m will eventually be delivered to all tier components $t_i \in t$ that are eventually functionally up and delivered some message $m' < m$ that has been produced before m by the same tier.* □

Note that this subtly also covers the case that m is committed to \mathcal{L} multiple times as we did neither specify the order of commit and delivery nor the log id with which m is delivered. We simply state that m is delivered, which might even be the case *before* m is committed a second, third or n th time.

PROOF (LEMMA 6.12) As for Lemma 6.11, if t_i does not crash, it follows from the properties of the replicated log or the respective consensus adapter that method `NEWENTRY` in algorithm 4.7 is invoked. Then, one out of four things can happen.

1. `unhandledEntries` is not empty. Then, as for Lemma 6.11, this leads to either 2., 3. or 4.
2. `unhandledEntries` is empty or `HANDLEENTRY` is invoked during recovery and m already has been delivered. Then, m will be discarded as `cond1` in Algorithm 4.7 is true, but we already fulfilled delivery by assumption.
3. `unhandledEntries` is empty or `HANDLEENTRY` is invoked during recovery and neither m nor message \tilde{m} with $\text{id}(\tilde{m}) = \text{id}(m) - 1$ and $\text{tierId}(\tilde{m}) = \text{tierId}(m)$, that is, the message that has been produced by the very same tier just before m , has been delivered. Then, condition `cond2` is true and m will be put into `unprocessedMessages` until its predecessor \tilde{m} has been delivered. By Lemma 6.9 and Algorithm 4.7 we can use induction over all preceding messages (ending with either the initial message or some message whose predecessor already has been delivered) to prove that this actually takes place eventually. Then, m will be delivered in the while loop of Algorithm 4.7 that is invoked when some $m' < m$ is delivered such that all messages up to \tilde{m} may be delivered, too.
4. `unhandledEntries` is empty or `HANDLEENTRY` is invoked during recovery and message \tilde{m} with $\text{id}(\tilde{m}) = \text{id}(m) - 1$ and $\text{tierId}(\tilde{m}) = \text{tierId}(m)$, that is, the message that has been produced by the very same tier just before m , has been delivered, but not yet m . Then, however, `cond2` (as well as `cond1`) in Algorithm 4.7 is false and m is applied to the state machine which equals delivery.

If t_i crashes before it delivered m , we can adopt the argument from Lemma 6.11 to show that `lastLogId` was for sure less than `logId(m)` by Algorithm 4.7. Following the argument, there will be some snapshot s from which t_i eventually succeeds to recover as it is assumed to be eventually functionally up. As snapshots are only taken during normal operation, it holds `logId(m) > s:lastLogId`. Then, however, m will be delivered during or after recovery. This is due to Algorithm 4.2, in which `unhandledEntries` is chosen just like that, and due to points 2., 3. or 4., respectively, from the argument we just developed for the non-crashing case. ■

Corollary 6.1 *If some tier t produces some message m_i , m_i will eventually be delivered to each consuming tier t_c whose `minACK` value at t is less or equal than i .* □

PROOF (COROLLARY 6.1) From Lemma 6.10, we know that the first message to be committed is the one whose `id` is the `minACK` value of t_c at t . From Lemma 6.11, we know that this first message will be for sure delivered. But then, we can apply Lemma 6.9 together with Lemma 6.12 for all other messages from this first message on. ■

Lemma 6.13 *If some tier t_c delivers a message m from tier t , tier t has a `minACK` value for t_c in its replicated log.* □

PROOF (LEMMA 6.13) As the channel may not create messages, some tier component $t_i \in t$ must have sent m . As sending only happens after connection establishment according to Algorithm 4.3, t_i will either be able to read an already existing minACK value from its log or it will try to commit an initial minACK value for t_c which succeeds unless t_i crashes before. However, t_i will neither send any message during connection establishment nor add t_c to the set T_c of locally known consuming tiers unless it succeeded in either reading or committing a minACK value for t_c . Adding t_c to T_c , though, is a necessary requirement to directly send a message to t_c according to Algorithm 4.8. ■

The lemmata above will turn out to be sufficient to prove suffix validity. The next lemma is necessary to prove agreement.

Lemma 6.14 *If some tier t sends a message m , m is produced by t 's state machine.* □

This lemma is like a *no creation* property for the outbound middleware.

PROOF (LEMMA 6.14) According to Algorithms 4.3 and 4.8, each tier only sends messages to its consumers which also are stored in *allMessages*. As *allMessages* is only filled by algorithm 4.8, whose SEND() method may only be invoked by the state machine, m must indeed have been produced by t 's state machine. ■

Lemma 6.15 *Each tier t delivers messages from one tier $t_p \in T_p$ in the order they were produced at t_p . That is, if m was produced by t_p before m' , m will be delivered before m' .* □

PROOF (LEMMA 6.15) The strict total order that is imposed on the outgoing messages is by Algorithm 4.8 stored in a message's id field $\text{id}(m)$. Furthermore, this id field is never changed after this point. Tier t only delivers m if its *lastProcessedMessage* value for t_c is $\text{id}(m) - 1$ according to Algorithm 4.7. By the strict total order of outgoing messages, this ensures delivery in precisely the order that messages were produced. ■

Lemma 6.16 *For each tier component $t_i \in t$ and each producing tier $t_p \in T_p$, the field $\text{lastProcessedMessage}[t_p]$ is monotonically increasing during normal operation* □

PROOF (LEMMA 6.16) During normal operation, the only place where *lastProcessedMessage* is changed, is in Algorithm 4.7. This algorithm, however, only changes it, if there is some m such that $\text{lastProcessedMessage}[\text{senderId}(m)] = \text{id}(m) - 1$. After this step, $\text{lastProcessedMessage}[\text{senderId}(m)]$ is set to $\text{id}(m)$. As all this is assumed to be done by exactly one thread (or multiple threads with adequate locking), this yields strict monotonicity of $\text{lastProcessedMessage}[t_p]$.

During recovery according to Algorithm 4.2, we process at least all entries that have been applied before the crash. As the order is the same, too, this will result in the very same changes of $\text{lastProcessedMessage}[t_p]$ that occurred before the crash. With the same monotonicity

argument as above, we also know that $lastProcessedMessage[t_p]$ does not decrease even if we process more entries than before the crash. However, as we process at least the same ordered set, we will reach at least the value that $lastProcessedMessage[t_p]$ had before the crash. This completes this proof for normal operation after recovery, too. ■

Lemma 6.17 *Each tier t delivers a message m at most once during normal operation.* □

PROOF (LEMMA 6.17) During normal operation, messages are only applied within Algorithm 4.7. This algorithm, however, only applies m if $lastProcessedMessage[senderId(m)] = id(m) - 1$. As $lastProcessedMessage[\cdot]$ is monotonically increasing by Lemma 6.16 and it is changed each time a message is applied, m must only be applied once during normal operation. ■

6.3. Proof of the Normal Operation Properties

Having proven these lemmata, we can now go on to prove the actual protocol properties we stated in chapter 3 for correct normal operation.

6.3.1. Suffix Validity

Let us quickly recall the definition of suffix validity:

If tier t produces some message $m' > m$ after some other message m and some tier $t_c \in T_c(m')$ delivered m , t_c will eventually deliver m' .

Theorem 6.1 (Suffix Validity) *The abstract implementation fulfills suffix validity.* □

PROOF (THEOREM 6.1) If tier t_c delivered some message m , it has some minACK value at tier t by Lemma 6.13. Then it will eventually deliver m' by Corollary 6.1. ■

6.3.2. Agreement

Agreement has been defined as:

If some tier $t_c \in T_c(m)$ delivers a message m , then every tier $t'_c \in T_c(m)$ will deliver m .

Theorem 6.2 (Agreement) *The abstract implementation fulfills agreement.* □

6. Proof

PROOF (THEOREM 6.2) As $T_c(m)$ is defined as the set of consuming tiers which already delivered some message m or one of its predecessors, by Lemma 6.13, each $t_c \in T_c(m)$ has a minACK entry with a value less or equal than $\text{id}(m)$ in the log which is replicated within tier $t : \text{tierId}(t) = \text{senderId}(m)$. Furthermore, as the channel may not create messages, m must have been sent by some tier component $t_i \in t$. By Lemma 6.14, m must have been produced by t . Then, however, we can apply Corollary 6.1 to all other consuming tiers $t'_c \in T_c(m)$, proving agreement. ■

6.3.3. Total Production Order

The property of total production order states:

If some tier t produces a message $m' > m$ after some other message m , every tier $t_c \in T_c(m')$ that delivers m' has to deliver m before it may deliver m' . Furthermore, every message m is delivered only once by each tier $t_c \in T_c(m)$. This is equivalent to $m' >^t m \Rightarrow m' >_i^{t_c} m$.

Theorem 6.3 (Total Production Order) *The abstract implementation fulfills total production order.* □

PROOF (THEOREM 6.3) This follows from Lemmata 6.15 and 6.17. ■

6.3.4. Integrity

Integrity is maybe the most simple property and has been defined as follows:

Every message m that is delivered by tier t is produced by exactly one preceding tier t_p .

Theorem 6.4 (Integrity) *The abstract implementation fulfills integrity.* □

PROOF (THEOREM 6.4) As we assumed channels without creation and messages are only stored, but not altered by a receiving tier, this follows from Lemma 6.14. ■

6.3.5. Initial Registration

Initial registration exists to ensure that in the case of a chain, all messages are processed. This has been formally stated by:

There is at least one tier $t_c \in T_c(m_0)$ that delivers the very first message m_0 produced by t .

Theorem 6.5 (Initial Registration) *The abstract implementation fulfills initial registration.* \square

PROOF (THEOREM 6.5) Firstly, by Algorithm 4.3, each tier t_c gets assigned a minACK value which is the minimum of all existing minACK values at this time.

As for one, first tier $t_{c,0}$ this assignment's condition will be fulfilled by the non-existence of any other minACK value, for $t_{c,0}$ the minACK value will be 0 by the definition of GETMINMINACK.

Furthermore, as *nextMessageId* only increases when a message is produced, but nowhere else – in particular neither by snapshotting nor recovery – and it is initially defined as 0, the very first message m_0 indeed will obtain id 0.

Then, we can combine Lemma 6.10 with Lemma 6.11 which state than the message with the minACK value will be eventually committed and – when it is committed – directly delivered to the state machine. \blacksquare

6.4. Proof of the Recovery Properties

The previous properties only handled the normal operation case. In this section, we will prove the two remaining properties, which are important for correct recovery.

6.4.1. Repeating Suffix Recovery

The repeating suffix recovery property guarantees that recovery will take place nicely and after recovery, the state machine will be in the same state it would have reached under normal operation, too.

If some tier component $t_i \in t$ recovers from some initial state or snapshot as it crashed before processing some message m_{crash} , the ordered set of messages *redelivered* at its state machine replica during recovery is an ordered suffix of the messages delivered by t until t_i crashed, $\{m : m \in IM \wedge m <_i m_{\text{crash}}\}$, ordered by the same total order $<_i$ these messages were delivered by before the crash.

Theorem 6.6 (Repeating Suffix Recovery) *The abstract implementation fulfills repeating suffix recovery.* \square

First, note, that recovery in this sense does not end when the implementation changes its mode to normal operation, but as soon as the suffix is applied. Everything after that is to be considered as normal operation.

PROOF (THEOREM 6.6) On recovery according to Algorithm 4.2, t_i reapplies all log entries that are more recent than the lastLogId of the snapshot it is recovering from. By the persistence of the replicated log, we know that all messages $m < m_{\text{crash}}$ have to be contained in the log. So we know that we pass at least all messages/entries $m : \text{logId}(m_{\text{crash}}) > \text{logId}(m) > \text{lastLogId}$ to HANDLEENTRY.

This, however, is just the same behavior as the one initiated by the replicated log during the normal operation case in Algorithm 4.7, inevitably leading to precisely the same order of delivery. ■

The idea which initially led to the implementation design could be used for this proof, too. It goes as follows: *lastProcessedMessage* and *unprocessedMessages* are – after the initial restore step during recovery – only changed by HANDLEENTRY which has to be either executed only by the state machine thread or with proper locking. Hence, we can then extend the state machine's state by those two variables and apply the state machine replication argument treating the recovered state machine as just another replica with HANDLEENTRY as the new function which applies entries to the extended state machine. As NEWENTRY will only be called with entries with a logId greater or equal than $\text{logId}(m_{\text{crash}})$ and *unhandledEntries* is processed in the order of logIds, this at least holds for all messages up to m_{crash} exclusively.

6.4.2. Message Identity

If some tier component $t_i \in t$ produces some message m' during recovery from some initial state or snapshot and this message is identical to another message m produced before it crashed – in particular identical regarding some identifier field – the production and delivery of m' and m is defined to be equivalent as they have the same effect on consuming tiers.

Theorem 6.7 (Message Identity) *The abstract implementation fulfills message identity.* □

What we need to show here is that some m' with $\text{id}(m') = i$ which is produced during recovery has the same effect as the message m with $\text{id}(m) = i = \text{id}(m')$ produced before the crash. Both messages are produced by the same tier, so their tier id is identical. To have a different effect, m and m' need to have different content. If the content was the same, they would not be distinguishable by a consuming tier t_c .

PROOF (THEOREM 6.7) So let us assume that $\text{id}(m) = \text{id}(m')$, but $\text{content}(m) \neq \text{content}(m')$.

As for the repeating suffix recovery idea, we can also extend the state machine's state by *nextMessageId*, which is consistently saved in and restored from snapshots. Then, assigning an id to a message or its content respectively becomes part of the still deterministic state change of a the extended state machine. As we recover from a consistent snapshot that includes *nextMessageId* and apply entries in the same order as they were applied between the snapshot we are recovering from and the crash – this is the same argument as for repeating suffix recovery – this state change will take place in the very same way it took place before the crash. In particular, the assignment of an id to some message content will be identical. This is a contradiction to our assumption, finally proving message identity. ■

7. Evaluation

To test our concrete implementation, we did some basic tests in a small scale environment. This should give a first impression of the performance of the protocol and our concrete implementation. Note that the replicated log is not optimized for this scenario nor is the concrete implementation optimized for performance. It is rather to be considered a proof of concept which additionally gives a rough estimate of the possible performance.

7.1. Setup

Our setup consists of several nodes that are interconnected using a local area network with a round-trip time of about 0.3 to 0.4 milliseconds between each pair of nodes. Each node has an Intel Xeon E3-1245v2 CPU with four cores running at 3.4 GHz, 16 GB of DDR3 memory and two Intel 520 solid state disks.

Figure 7.1 shows the structure of our test setup. A chain of tiers with variable length is connected to a variable number of nodes that act as sources and sinks and thus are able to measure throughput and latency.

Each tier consists of three nodes which replicate a state machine which simply relays all messages unchanged. Thus, it has no distinct state besides the replicated log and the necessary meta information described in the implementation chapters. Separate evaluations showed that the bottleneck of the current implementation is the replicated log. It contributes the most to the overall latency by replicating each message to at least a majority of the tier components and limits the throughput by writing all messages to stable storage.

When we increase the number of tiers in the chain, we will call this horizontal scaling. Increasing the number of source/sink nodes will be called vertical scaling. This naming is in accordance with Figure 7.1.

7.2. Behavior under Load

To get a rough impression how many messages should be sent in parallel to saturate all buffers and delaying elements, the first benchmark evaluates only one tier together with one source/sink node over a variable number of so-called in-flight messages. This means, the

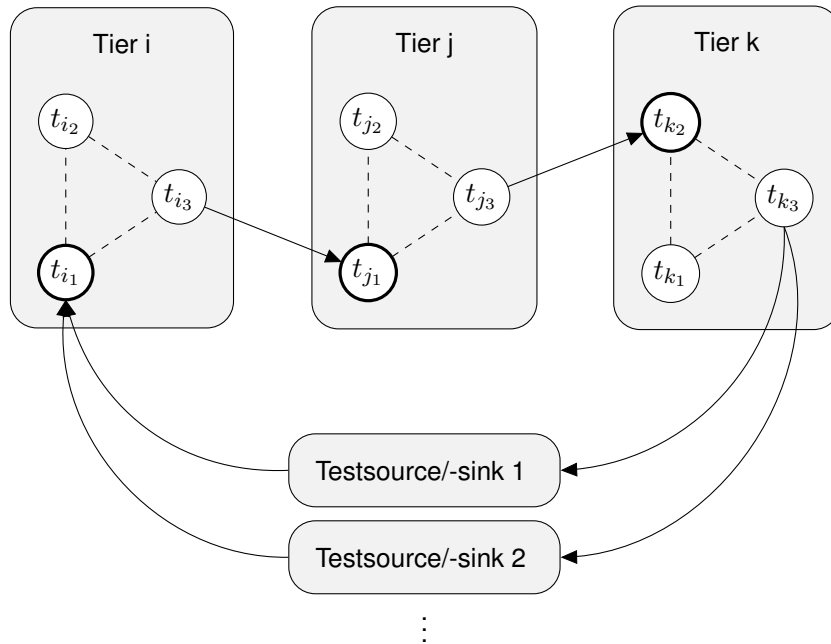


Figure 7.1.: The benchmarking setup. A chain of tiers of variable length is at each end connected with a variable number of nodes that act as both, source and sink. Thereby, it is possible to measure throughput and latency of the chain.

source/sink node generates this specific number of messages, sends them all and then waits for the respective message to be passed through the tier that we want to benchmark. For each message that is returned, the source/sink node may generate a new one such that the in-flight messages can be imagined either on the way to, on the way from or inside the tier we are evaluating with their total number being kept constant.

The results are depicted in Figure 7.2. For up to 20 in-flight messages, the throughput increases. Then, it flattens at about 7000 messages per second. At the same time, the latency stays below 3 milliseconds for up to 20 in-flight messages, but is already greater than 1 millisecond for more than one message. This gives us 20 in-flight messages as a rough estimate, which we use for the next benchmarks, to represent a good tradeoff between throughput and latency.

7.3. Vertical Scaling

In the vertical scaling scenario, we still only consider one tier. However, we now increase the number of source/sink nodes from one to up to four nodes. Again, we measure throughput and latency and the results can be found in Figure 7.3.

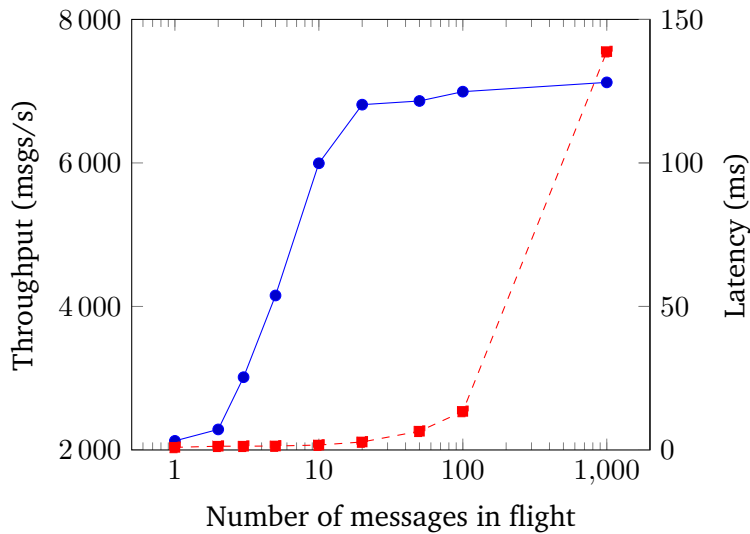


Figure 7.2.: To estimate the behavior of a single tier under load, a varying number of messages is being sent in parallel. With the number of so-called in-flight messages, throughput (blue/solid) and latency (red/dashed) increase.

We see that the latency stays almost constant and thus can be assumed to be independent of the number of source/sink nodes. While throughput stays constant for up to three source/sink nodes, too, it drops by about 10% as soon as we extend the scenario and connect a fourth source/sink node. This could be explained by the fact that we distributed the test nodes in their roles as sinks over all three tier components, but when a fourth node is connected, one tier component has to provide two consumers with all messages. This assumption is supported by the fact that we observed similar behavior as soon as we connected all three source/sink to only one tier component.

7.4. Horizontal Scaling

Finally, we also considered one scenario with a chain length of two, which we call horizontal scaling. While the results in figure 7.4 show the numbers for two source/sink nodes, the results did not significantly differ for only one source/sink node.

With respect to latency, the results are as the benchmark scenario would suggest. Latency almost exactly doubles which can be explained by the fact that each tier has to replicate each message internally. With respect to throughput, however, we did not expect such a significant depletion. We suppose that this results from internal delays which are induced by the increased latency.

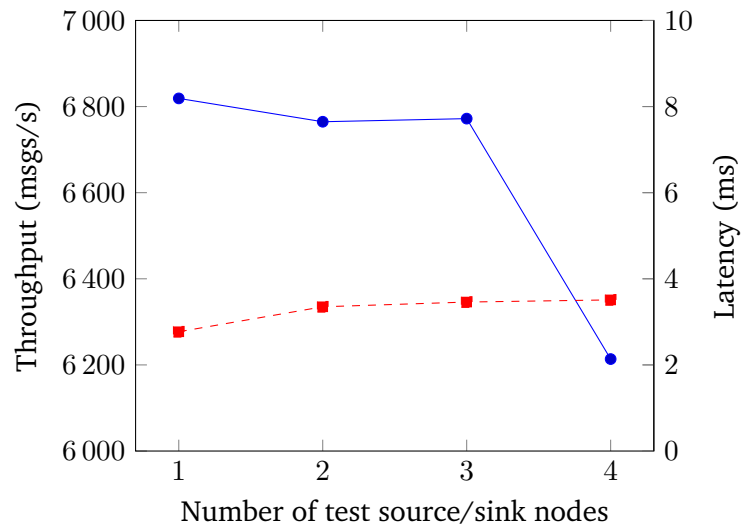


Figure 7.3.: To benchmark a tier for a varying number of sources and sinks, in this vertical scaling scenario, we measure throughput (blue/solid) and latency (red/dashed). While latency is fairly constant, throughput drops when a fourth source/sink is connected.

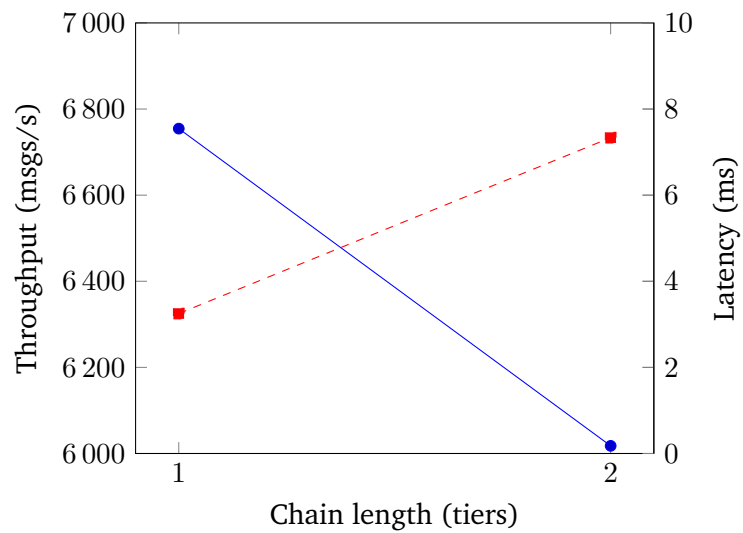


Figure 7.4.: The length of a chain not only increases latency (red/dashed), but also decreases throughput (blue/solid) by a factor larger than expected.

8. Conclusion and Outlook

In our work, we have introduced a protocol to compose a generic highly available service network by consistently connecting components of a single service that use state machine replication to achieve high availability. While we provide high availability using multiple replicas of which a minority may fail, we still guarantee strong, sequential consistency. This approach comes at the cost of more replicas and only moderate throughput compared to stream processing systems.

The main ideas are an intelligent replicated log that provides very specific guarantees on the order of entries that are replicated and a persistent enumeration system that provides unique identifiers to enable safe recovery and replay of messages at both, the inbound and outbound side of the state machine.

Our current implementation is only semi-dynamic as new tiers can only be added, but not removed from the network. This could be changed in future work once more leveraging the guarantees the replicated log may provide.

We could also imagine to extend the failure model from a crash-recovery to a Byzantine failure model by extending the protocol in the ways already known for replication protocols [Lis10].

Other included or missing features result from design decisions. Firstly, we dropped causal consistency in favor of an arbitrary network topology that also may contain cycles. While we cannot prove that both are mutually exclusive, all attempts to implement it failed as soon as the network could contain a cycle. Secondly, we decided to use stable storage instead of recovering the state from other replicas as in Viewstamped Replication as our replicated log required stable storage anyway. Finally, the instability of the existing Raft implementations in Java lead us to the decision to do the message filtering close to the state machine instead of implementing the filter inside the replicated log.

All in all, we think that our work has shown that generic network topologies known from stream processing systems can be combined with the sequential consistency and high availability guarantees known from database and three-tier architecture systems. At the same time, we encapsulated recovery and filtering mechanisms inside the single tiers such that tiers from different vendors or service providers can easily be combined as soon as they follow the very simple protocol that we proposed.

A. A Simple Implementation of an “Intelligent” Replicated Log

Many parts of the abstract implementation can be simplified by using a replicated log that gives the implementer guarantees about which entries are committed, which entries are discarded or which entries are committed, but hidden from the application built around it. Examples are the ability to commit an entry together with a condition and this entry only is replicated if the condition is fulfilled, the guarantee of some user-specified order or the deduplication of entries based on some user-specified criterion like an id.

With conditional or transactional commits, ZooKeeper’s internal log already supports these examples. ZooKeeper provides these features – to our best, but still limited understanding of its internals – within the leader, which does not propose commits that do not pass transactionally defined checks or do not meet previously assumed versions of a value.

While it is the probably most performant way to modify the code of the replicated log which determines which requests a leader actually proposes, this might not always be possible. One might, however, easily implement a similarly intelligent replicated log by modifying the read functions or filtering the invocations that are done by the log. This will now be explained in detail. Thereby, we assume that we can influence the entries that are committed to the log and modify the data that leaves the log. This could be done by a wrapper similar to the already proposed consensus adapter.

Example 1: Conditional Commits

A conditional commit can impose a condition *cond* on the log entries $\tilde{e} \in \mathcal{L} : \text{logId}(\tilde{e}) < \text{logId}(e)$ that exist before the associated entry *e* is committed to the log. It must only be delivered to the surrounding application if *cond* is met.

Instead of committing only an entry *e*, one extends this to $e' = (e, \text{cond})$. If then *e'* is committed, it can only be read or the wrapper will only notify the surrounding application of a new entry in case the condition is fulfilled by the log entries $\tilde{e} \in \mathcal{L} : \text{logId}(\tilde{e}) < \text{logId}(e)$. While this might lead to commits that are ignored forever, it perfectly fits the requirements.

Note that we did not specify how the application can determine whether it can confirm the commit or not. This, however, can simply be done if the entry is uniquely identifiable – in case

there is no identifier yet, this could be done by extending the entry again. Then, the application reads the log via the wrapper after the commit returned successfully. If it may retrieve the log entry it was supposed to commit, it can confirm the commit, otherwise it has to deny it.

Example 2: Deduplication of Entries

This is a simple special case of example 1 where the condition is that an entry with a specific name or id is not yet existent. However, the application may confirm all duplicate commits or even give the more detailed feedback that it is a duplicate.

Example 3: Order by Some $\text{id}(e)$

In this case, the log should only commit e with $\text{id}(e) = i$ if there is an entry e' with $\text{id}(e') = i - 1$. The easiest way would be to make this a condition and proceed like in example 1. However, there is a more elegant and still deterministic way to do this.

Assume that entries are now specified by their ids. Then, assume that entries 2 and 3 are already committed, but – like for example 1 – not readable yet. If the missing entry 1 is then committed, we can simply specify that now all three entries are readable and they are ordered in the log directly after entry 1, that is, in particular before all other entries committed after entry 1. This yields a deterministic order which is available faster to the application, possibly reducing latency.

As, by the initial idea, we still have to deny the commits for entry 2 and 3, they will eventually be committed again. These entries then will be ignored as duplicates in example 2 are ignored. The application may either confirm the commit or provide some more detailed result. A more elegant solution would be to defer the acknowledgments for entries 2 and 3 and reply with a confirmation as soon as entry 1 is committed.

Bibliography

- [Aba10] Daniel Abadi. *Problems with CAP, and Yahoo’s little known NoSQL system*. 2010. URL: <http://dbmsmusings.blogspot.de/2010/04/problems-with-cap-and-yahoos-little.html> (cit. on p. 13).
- [ACT00] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. “Failure detection and consensus in the crash-recovery model.” In: *Distributed Computing* 13.2 (2000), pp. 99–125. ISSN: 0178-2770. DOI: [10.1007/s004460050070](https://doi.org/10.1007/s004460050070) (cit. on p. 24).
- [AW04] Hagit Attiya and Jennifer Welch. *Distributed computing: Fundamentals, simulations, and advanced topics*. 2nd ed. Wiley series on parallel and distributed computing. Hoboken, NJ: Wiley, 2004. ISBN: 978-0-471-45324-6 (cit. on p. 16).
- [Bai+13] Peter Bailis et al. “HAT, Not CAP: Towards Highly Available Transactions.” In: *Presented as part of the 14th Workshop on Hot Topics in Operating Systems*. Berkeley, CA: USENIX, 2013. URL: <https://www.usenix.org/conference/hotos13/hat-not-cap-highly-available-transactions> (cit. on p. 13).
- [BFF09a] A. Brito, C. Fetzer, and P. Felber. “Minimizing Latency in Fault-Tolerant Distributed Stream Processing Systems.” In: *2009 29th IEEE International Conference on Distributed Computing Systems (ICDCS)*. 2009, pp. 173–182. DOI: [10.1109/ICDCS.2009.35](https://doi.org/10.1109/ICDCS.2009.35) (cit. on p. 21).
- [BFF09b] Andrey Brito, Christof Fetzer, and Pascal Felber. “Multithreading-Enabled Active Replication for Event Stream Processing Operators.” In: *2009 28th IEEE International Symposium on Reliable Distributed Systems (SRDS)*. 2009, pp. 22–31. DOI: [10.1109/SRDS.2009.37](https://doi.org/10.1109/SRDS.2009.37) (cit. on p. 21).
- [BG05] Romain Boichat and Rachid Guerraoui. “Reliable and total order broadcast in the crash-recovery model.” In: *Journal of Parallel and Distributed Computing* 65.4 (2005), pp. 397–413. ISSN: 07437315. DOI: [10.1016/j.jpdc.2004.10.008](https://doi.org/10.1016/j.jpdc.2004.10.008) (cit. on p. 24).
- [BKL20] Bela Ban, Ensar Basri Kagveci, and Ugo Landini. *jgroups-raft*. 8.7.2015. URL: <https://github.com/belaban/jgroups-raft> (cit. on p. 55).
- [Bre00] Eric A. Brewer. “Towards robust distributed systems.” In: *the nineteenth annual ACM symposium*. Ed. by Gil Neiger. 2000, p. 7. DOI: [10.1145/343477.343502](https://doi.org/10.1145/343477.343502) (cit. on pp. 11, 13).

- [Bur06] Mike Burrows. “The Chubby Lock Service for Loosely-coupled Distributed Systems.” In: *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*. OSDI ’06. Berkeley, CA, USA: USENIX Association, 2006, pp. 335–350. ISBN: 1-931971-47-1. URL: <http://dl.acm.org/citation.cfm?id=1298455.1298487> (cit. on pp. 16, 53).
- [CDK05] George F. Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: Concepts and design*. 4th ed. International computer science series. Harlow, England and New York: Addison-Wesley, 2005. ISBN: 0321263545 (cit. on pp. 11, 12, 16, 18).
- [CGR07] Tushar D. Chandra, Robert Griesemer, and Joshua Redstone. “Paxos Made Live: An Engineering Perspective.” In: *the twenty-sixth annual ACM symposium*. Ed. by Indranil Gupta and Rogert Wattenhofer. 2007, pp. 398–407. DOI: [10.1145/1281100.1281103](https://doi.org/10.1145/1281100.1281103) (cit. on pp. 16, 53).
- [Cor+13] James C. Corbett et al. “Spanner: Google’s Globally Distributed Database.” In: *ACM Trans. Comput. Syst.* 31.3 (2013), 8:1–8:22. ISSN: 0734-2071. DOI: [10.1145/2491245](https://doi.org/10.1145/2491245). URL: <http://doi.acm.org/10.1145/2491245> (cit. on pp. 16, 53).
- [Cor08] Inc CoreOS. *etcd*. 28.08.2015. URL: <https://github.com/coreos/etcd> (cit. on p. 54).
- [CT96] Tushar Deepak Chandra and Sam Toueg. “Unreliable failure detectors for reliable distributed systems.” In: *Journal of the ACM* 43.2 (1996), pp. 225–267. ISSN: 00045411. DOI: [10.1145/226643.226647](https://doi.org/10.1145/226643.226647) (cit. on pp. 15, 16, 24).
- [DSU04] Xavier Défago, André Schiper, and Péter Urbán. “Total order broadcast and multicast algorithms.” In: *ACM Computing Surveys* 36.4 (2004), pp. 372–421. ISSN: 03600300. DOI: [10.1145/1041680.1041682](https://doi.org/10.1145/1041680.1041682) (cit. on p. 18).
- [FG00] S. Frolund and R. Guerraoui. “Implementing e-Transactions with asynchronous replication.” In: *International Conference on Dependable Systems and Networks (includes FTCS-30 30th Annual International Symposium on Fault-Tolerant Computing and DCCA-8)*. 2000, pp. 449–458. DOI: [10.1109/ICDSN.2000.857575](https://doi.org/10.1109/ICDSN.2000.857575) (cit. on p. 20).
- [FG01] Svend Frølund and Rachid Guerraoui. “X-Ability: a theory of replication.” In: *Distributed Computing* 14.4 (2001), pp. 231–249. ISSN: 0178-2770. DOI: [10.1007/s004460100065](https://doi.org/10.1007/s004460100065) (cit. on p. 21).
- [FG02] S. Frolund and R. Guerraoui. “e-Transactions: end-to-end reliability for three-tier architectures.” In: *IEEE Transactions on Software Engineering* 28.4 (2002), pp. 378–395. ISSN: 0098-5589. DOI: [10.1109/TSE.2002.995430](https://doi.org/10.1109/TSE.2002.995430) (cit. on p. 20).
- [FLP85] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. “Impossibility of distributed consensus with one faulty process.” In: *Journal of the ACM* 32.2 (1985), pp. 374–382. ISSN: 00045411. DOI: [10.1145/3149.214121](https://doi.org/10.1145/3149.214121) (cit. on p. 16).

- [Geo14] Allen George. *Libraft*. 10.04.2014. URL: <https://github.com/allengeorge/libraft> (cit. on p. 54).
- [GK02] Guanling Chen and D. Kotz. “Context aggregation and dissemination in ubiquitous computing systems.” In: *Fourth IEEE Workshop on Mobile Computing Systems and Applications*. 2002, pp. 105–114. DOI: [10.1109/MCSA.2002.1017490](https://doi.org/10.1109/MCSA.2002.1017490) (cit. on p. 11).
- [GL02] Seth Gilbert and Nancy Lynch. “Brewer’s conjecture and the feasibility of consistent, available, partition-tolerant web services.” In: *ACM SIGACT News* 33.2 (2002), p. 51. ISSN: 01635700. DOI: [10.1145/564585.564601](https://doi.org/10.1145/564585.564601) (cit. on pp. 11, 13).
- [GOS98] Rachid Guerraoui, Riucarlos Oliveira, and André Schiper. *Stubborn communication channels*. Tech. rep. 98/009. Switzerland: Département d’Informatique, Ecole de Polytechnique Fédérale, 1998 (cit. on pp. 24, 67).
- [Hei+13] Thomas Heinze et al. “Fault-tolerant complex event processing using customizable state machine-based operators.” In: *the 15th International Conference*. Ed. by Elke Rundensteiner et al. 2013, pp. 590–593. DOI: [10.1145/2247596.2247673](https://doi.org/10.1145/2247596.2247673) (cit. on p. 21).
- [HH08] Jordan Halterman and Jonathan Halterman. *Copycat: Persistent - Consistent - Fault-tolerant - Database - Coordination - Framework*. 31.08.2015. URL: <https://github.com/kuujo/copycat> (cit. on p. 54).
- [Hsi04] Jonathan Hsieh. [*ZOOKEEPER-1162*] *consistent handling of jute.maxbuffer when attempting to read large zk "directories" - ASF JIRA*. 11.04.2015. URL: <https://issues.apache.org/jira/browse/ZOOKEEPER-1162> (cit. on p. 57).
- [Hun+10] Patrick Hunt et al. “ZooKeeper: Wait-free Coordination for Internet-scale Systems.” In: *USENIX Annual Technical Conference*. Vol. 8. 2010, p. 9 (cit. on p. 19).
- [HW90] Maurice P. Herlihy and Jeannette M. Wing. “Linearizability: a correctness condition for concurrent objects.” In: *ACM Transactions on Programming Languages and Systems* 12.3 (1990), pp. 463–492. ISSN: 01640925. DOI: [10.1145/78969.78972](https://doi.org/10.1145/78969.78972) (cit. on p. 12).
- [JG] Red Hat. *jgroups*. URL: <http://www.jgroups.org/> (cit. on p. 55).
- [JRS11] F. P. Junqueira, B. C. Reed, and M. Serafini. “Zab: High-performance broadcast for primary-backup systems.” In: *2011 IEEE/IFIP 41st International Conference on Dependable Systems & Networks (DSN)*. 2011, pp. 245–256. DOI: [10.1109/DSN.2011.5958223](https://doi.org/10.1109/DSN.2011.5958223) (cit. on pp. 9, 18, 19).
- [JS13] Flavio P. Junqueira and Marco Serafini. “On Barriers and the Gap between Active and Passive Replication.” In: *Distributed Computing*. Ed. by David Hutchison et al. Vol. 8205. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 299–313. ISBN: 978-3-642-41526-5. DOI: [10.1007/978-3-642-41527-2_21](https://doi.org/10.1007/978-3-642-41527-2_21) (cit. on p. 17).

- [Kol04] Heine Kolltveit. “Techniques for Achieving Exactly-Once Execution Semantics and High Availability for Multi-Tier Applications.” PhD thesis. Trondheim: Norwegian University of Science and Technology, 2004. URL: <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.60.4391&rep=rep1&type=pdf> (cit. on p. 21).
- [Lam01] Leslie Lamport. *Paxos Made Simple*. Ed. by Sergio Rajsbaum. New York, NY, USA, 2001. URL: <http://research.microsoft.com/en-us/um/people/lamport/pubs/paxos-simple.pdf> (cit. on p. 16).
- [Lam78] Leslie Lamport. “Time, clocks, and the ordering of events in a distributed system.” In: *Communications of the ACM* 21.7 (1978), pp. 558–565. ISSN: 00010782. DOI: [10.1145/359545.359563](https://doi.org/10.1145/359545.359563) (cit. on pp. 9, 14).
- [Lam79] Lamport. “How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs.” In: *IEEE Transactions on Computers* C-28.9 (1979), pp. 690–691. ISSN: 0018-9340. DOI: [10.1109/TC.1979.1675439](https://doi.org/10.1109/TC.1979.1675439) (cit. on p. 12).
- [Lam98] Leslie Lamport. “The part-time parliament.” In: *ACM Transactions on Computer Systems* 16.2 (1998), pp. 133–169. ISSN: 07342071. DOI: [10.1145/279227.279229](https://doi.org/10.1145/279227.279229) (cit. on pp. 9, 16).
- [LC12] Barbara Liskov and James Cowling. “Viewstamped Replication Revisited.” In: *CSAIL Technical Reports (July 1, 2003 - present)*. Vol. MIT-CSAIL-TR-2012-021. 2012. URL: <http://hdl.handle.net/1721.1/71763> (cit. on pp. 18, 36).
- [Lis10] Barbara Liskov. “From Viewstamped Replication to Byzantine Fault Tolerance.” In: *Replication*. Ed. by Bernadette Charron-Bost, Fernando Pedone, and André Schiper. Vol. 5959. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2010, pp. 121–149. ISBN: 978-3-642-11293-5. DOI: [10.1007/978-3-642-11294-2_7](https://doi.org/10.1007/978-3-642-11294-2_7). URL: http://dx.doi.org/10.1007/978-3-642-11294-2_7 (cit. on pp. 18, 81).
- [LMS11] Mikel Larrea, Cristian Martín, and Iratxe Soraluze. “Communication-efficient leader election in crash–recovery systems.” In: *Journal of Systems and Software* 84.12 (2011), pp. 2186–2195. ISSN: 01641212. DOI: [10.1016/j.jss.2011.06.019](https://doi.org/10.1016/j.jss.2011.06.019) (cit. on p. 20).
- [LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. “The Byzantine Generals Problem.” In: *ACM Transactions on Programming Languages and Systems* 4.3 (1982), pp. 382–401. ISSN: 01640925. DOI: [10.1145/357172.357176](https://doi.org/10.1145/357172.357176) (cit. on p. 16).
- [Mar+11] Andre Martin et al. “Low-Overhead Fault Tolerance for High-Throughput Data Processing Systems.” In: *2011 31st International Conference on Distributed Computing Systems (ICDCS)*. 2011, pp. 689–699. DOI: [10.1109/ICDCS.2011.29](https://doi.org/10.1109/ICDCS.2011.29) (cit. on p. 21).

- [Maz96] Karim Riad Mazouni. “Étude de l’invocation entre objets dupliqués dans un système réparti tolérant aux fautes.” PhD thesis. École polytechnique fédérale de Lausanne: Lausanne, 1996. URL: <http://infoscience.epfl.ch/record/32055> (cit. on p. 20).
- [MGG95a] K. R. Mazouni, B. Garbinato, and R. Guerraoui. “Filtering duplicated invocations using symmetric proxies.” In: *International Workshop on Object Orientation in Operating Systems*. 1995, pp. 118–126. DOI: [10.1109/IW00S.1995.470570](https://doi.org/10.1109/IW00S.1995.470570) (cit. on p. 20).
- [MGG95b] Karim Riad Mazouni, Benoît Garbinato, and Rachid Guerraoui. *Invocation Support for Replicated Objects*. 1995 (cit. on p. 20).
- [OL88] Brian M. Oki and Barbara H. Liskov. “Viewstamped replication: a general primary copy.” In: *the seventh annual ACM Symposium*. Ed. by Danny Dolev. 1988, pp. 8–17. DOI: [10.1145/62546.62549](https://doi.org/10.1145/62546.62549) (cit. on pp. 9, 18).
- [Ong08] Diego Ongaro. *The Raft Consensus Algorithm*. 27.08.2015. URL: <https://raft.github.io/> (cit. on p. 54).
- [OO14a] Diego Ongaro and John Ousterhout. “In Search of an Understandable Consensus Algorithm.” In: *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. Philadelphia, PA: USENIX Association, 2014, pp. 305–319. ISBN: 978-1-931971-10-2. URL: <https://www.usenix.org/conference/atc14/technical-sessions/presentation/ongaro> (cit. on pp. 9, 17).
- [OO14b] Diego Ongaro and John Ousterhout. *In Search of an Understandable Consensus Algorithm (Extended Version)*. 2014. URL: <https://ramcloud.stanford.edu/raft.pdf> (cit. on p. 17).
- [Pap03] M. P. Papazoglou. “Service-oriented computing: concepts, characteristics and directions.” In: *Fourth International Conference on Web Information Systems Engineering*. 2003, pp. 3–12. DOI: [10.1109/WISE.2003.1254461](https://doi.org/10.1109/WISE.2003.1254461) (cit. on p. 11).
- [PKS03a] S. Pleisch, A. Kupsys, and A. Schiper. “Preventing orphan requests in the context of replicated invocation.” In: *22nd International Symposium on Reliable Distributed Systems*. 2003, pp. 119–128. DOI: [10.1109/RELDIS.2003.1238061](https://doi.org/10.1109/RELDIS.2003.1238061) (cit. on p. 20).
- [PKS03b] Stefan Pleisch, Arnas Kupsys, and André Schiper. *Replicated Invocation*. Lausanne, 2003. URL: <http://infoscience.epfl.ch/record/52515> (cit. on p. 20).
- [PSL80] M. Pease, R. Shostak, and L. Lamport. “Reaching Agreement in the Presence of Faults.” In: *Journal of the ACM* 27.2 (1980), pp. 228–234. ISSN: 00045411. DOI: [10.1145/322186.322188](https://doi.org/10.1145/322186.322188) (cit. on p. 16).
- [RJ08] Benjamin Reed and Flavio P. Junqueira. “A simple totally ordered broadcast protocol.” In: *the 2nd Workshop*. Ed. by Eliezer Dekel and Gregory Chockler. 2008, p. 1. DOI: [10.1145/1529974.1529978](https://doi.org/10.1145/1529974.1529978) (cit. on p. 19).

- [RSS15] Robbert van Renesse, Nicolas Schiper, and Fred B. Schneider. “Vive La Différence: Paxos vs. Viewstamped Replication vs. Zab.” In: *IEEE Transactions on Dependable and Secure Computing* 12.4 (2015), pp. 472–484. ISSN: 1545-5971. DOI: [10.1109/TDSC.2014.2355848](https://doi.org/10.1109/TDSC.2014.2355848) (cit. on pp. 18, 19, 54).
- [Sch90] Fred B. Schneider. “Implementing fault-tolerant services using the state machine approach: a tutorial.” In: *ACM Computing Surveys* 22.4 (1990), pp. 299–319. ISSN: 03600300. DOI: [10.1145/98163.98167](https://doi.org/10.1145/98163.98167) (cit. on pp. 9, 14).
- [VS14] Al Vermeulen and Swami Sivasubramanian. *Under the Covers of AWS: Core Distributed Systems Primitives That Power Our Platform*. Ed. by AWS re:Invent 2014. Las Vegas, 2014. URL: <http://de.slideshare.net/AmazonWebServices/spot302-under-the-covers-of-aws-core-distributed-systems-primitives-that-power-our-platform-aws-reinvent-2014> (cit. on p. 16).
- [Wik15] Wikipedia. *Paxos (computer science)* — *Wikipedia, The Free Encyclopedia*. [Online; accessed 22-September-2015]. 2015. URL: [https://en.wikipedia.org/w/index.php?title=Paxos_\(computer_science\)&oldid=679752787](https://en.wikipedia.org/w/index.php?title=Paxos_(computer_science)&oldid=679752787) (cit. on p. 16).
- [WK08] Huaigu Wu and Bettina Kemme. “Showing correctness of a replication algorithm in a component based system.” In: *the 2008 international symposium*. Ed. by Bipin C. Desai. 2008, p. 91. DOI: [10.1145/1451940.1451954](https://doi.org/10.1145/1451940.1451954) (cit. on p. 21).
- [ZK] The Apache Software Foundation. *Apache ZooKeeper*. URL: <http://zookeeper.apache.org/> (cit. on p. 55).
- [ZKAG] The Apache Software Foundation. *ZooKeeper Administrator’s Guide: A Guide to Deployment and Administration*. 13.03.2014. URL: <http://zookeeper.apache.org/doc/r3.4.6/zookeeperAdmin.html> (cit. on p. 60).
- [ZMMS02] W. Zhao, L. E. Moser, and P. M. Melliar-Smith. “Unification of replication and transaction processing in three-tier architectures.” In: *22nd International Conference on Distributed Computing Systems*. 2002, pp. 290–297. DOI: [10.1109/ICDCS.2002.1022266](https://doi.org/10.1109/ICDCS.2002.1022266) (cit. on p. 21).

Declaration

I hereby declare that the work presented in this thesis is entirely my own. I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature