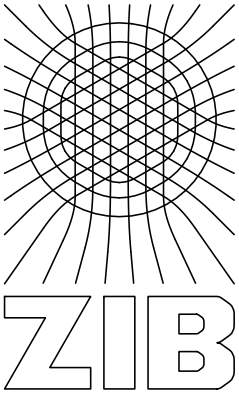
Zuse Institute Berlin

Takustr. 7
14195 Berlin
Germany



JAN SKRZYPCZAK

Weakening Paxos Consensus Sequences for Commutative Commands

This work was submitted and accepted as master's thesis at Humboldt University of Berlin in partial fulfilment of the requirements for the degree of Master of Science.

ZIB Report 17-64 (November 2017)

Zuse Institute Berlin
Takustr. 7
14195 Berlin
Germany

Telephone: +49 30-84185-0
Telefax: +49 30-84185-125

E-mail: bibliothek@zib.de
URL: http://www.zib.de

ZIB-Report (Print) ISSN 1438-0064
ZIB-Report (Internet) ISSN 2192-7782

Abstract

Consensus (agreement on a value) is regarded as a fundamental primitive in the design of fault tolerant distributed systems. A well-known solution to the consensus problem is Paxos. Extensions of the Paxos algorithm make it possible to reach agreement on a sequence of commands which can then be applied on a replicated state. However, concurrently proposed commands can create conflicts that must be resolved by ordering them.

This thesis delivers an in-depth description of a Paxos-based algorithm to establish such command sequences, called Paxos Round Based Register (PRBR). In contrast to conventional approaches like Multi-Paxos, PRBR can manage multiple command sequences independently. Furthermore, each sequence is established in-place, which eliminates the need for managing multiple Paxos instances.

PRBR is extended as part of this thesis to exploit the commutativity of concurrently proposed commands. As a result, conflict potential can be greatly reduced which increases the number of commands that can be handled by PRBR. This is shown for a number of workloads in an experimental evaluation.

Contents

List of Figures

List of Execution Examples

List of Code Listings

1 Introduction

1.1 Motivation

The consensus problem is one of the most fundamental challenges in distributed systems. In its simplest form, it requires a number of processes to agree on a single data value in spite of possible process failures and unreliable communication links. A number of approaches that solve the consensus problem exist – one of them being the Paxos algorithm.

The basic Paxos algorithm can be extended to achieve consensus on a sequence of values. Such a consensus sequence can be used as a component in the design of a replicated data storage. Clients submit commands that operate on the replicated data, which are then ordered by the algorithm and applied on each replica in the same order.

Traditional approaches, for example Multi-Paxos, establish a total order of the submitted commands over all replicas. Conflicts can occur if commands are submitted concurrently. These have to be resolved first before the respective commands can be applied. Naturally, any conflict resolution mechanism introduces additional computational and communication overhead. Thus, a large number of conflicts negatively affects the system's performance.

A strict ordering of commands is often not necessary. Often, a large set of independent data objects is managed by the storage system. Two commands that modify or access different objects do not interfere with the result of each other. In other words, they commute. Other examples of commutative commands include commands that only read the state of data objects or commands that modify independent parts of a more complexly structured object.

The result of commutative commands is independent of their execution order. Therefore, replicas might apply them in arbitrary order without introducing inconsistencies to the systems. The commutative properties of commands can be exploited by the consensus algorithm to reduce the overhead required ordering them, which in turn should have a positive impact on the system's performance.

1.2 Contribution

The contribution of this thesis is twofold. First, providing the first textual description of PRBR (Paxos Round Based Register) and second, extending PRBR for making use of commutative commands.

Documentation of PRBR PRBR is a Paxos-based protocol with the capabilities of establishing a register of consensus sequences. Development of PRBR started end of 2012 in the context of Scalaris, a distributed key-value store, which is an ongoing research project at Zuse Institue Berlin. No textual documentation of PRBR's current state exists. Chapter 7 provides a detailed description of PRBR, along with a number of informal proof sketches to give an intuition about PRBR's correctness. In addition, a number of minor modifications were made to PRBR in the preparation of this thesis.

Exploiting Commutative Commands A key property of PRBR is it to establish independent consensus sequences for separate data objects. By doing so, it can establish consensus of commands targeting different objects without ordering them. However, a strict sequential order of commands targeting the same object is still required. The main contribution of this thesis constitutes the extension of PRBR to weaken the ordering constraints within a consensus sequence by exploiting the commutative relationships between submitted commands. These modifications are discussed in Chapter 8. Afterwards, the resulting protocol is evaluated by comparing its performance to PRBR's unmodified state under different workloads in Chapter 10.

Part I
Background and Related Work

2 Distributed System Abstraction

A typical distributed system consists of multiple physical machines that interact with each other over a communication network to achieve some common goal. Depending on the scale of such system, the number of machines involved can be large. Each machine has the potential to fail at any given time since no hardware or software of a non-trivial complexity is free of fault. In addition, failures of the communication network must be considered as well. Information transferred via the network can get lost, duplicated, corrupted or simply delayed for a long period of time before reaching its destination. Therefore, it is important to design a distributed system in a way that is resilient to failures of different kinds.

A distributed algorithm is an algorithm designed to run on independent, interconnected components, which are typically (but not necessarily) spread across multiple physical machines. The behavior of such an algorithm depends strongly on the environment it is used in. The manner components act, how they communicate with each other, and what types of failures are possible are only some of the factors that must be considered. To make a distributed algorithm transferable, using system abstractions is important to provide certain guarantees about the behavior of a distributed system.

2.1 Processes

In the context of this thesis, a process is considered to be an abstract unit that is able to perform computations. A distributed system consists of a number of different processes, which can communicate with each other by exchanging messages via communication links (further specified in Section 2.2). Every process has a *process id* (PID), which is unique across all processes in the system. A PID is sufficient to identify its corresponding process in the system and can be used to send messages to it.

A physical machine can contain an arbitrary number of processes. It is important to note that the notion of a process as described here is independent to actual processes used by an operating system or runtime environment. Unless stated otherwise, no particular mapping between the two concepts is assumed.

A computation which can be executed by a process is called an *action*. A process must fulfill certain requirements before it can execute one of its actions. If those requirements are fulfilled, the action is called *enabled*. The requirements of an action are typically that a message, or a set of messages with certain content must have been received by the process containing this action. Once an action is enabled, it is executed immediately. Actions can be of arbitrary complexity and can contain any kind of instructions, including sending messages to other processes.

In the context of this thesis, however, processes are not allowed to receive messages during the execution of an action. That way, a process can only execute at most one action at any given time. The set of messages expected by a process (and by extension the actions that can be triggered by these messages) are called its *interface*.

During normal operation, every process p executes the following loop: The process waits for an incoming message to receive. Once p receives a message, it checks if its arrival enables an action. If an action is enabled p executes it immediately. After p completed all computations, it waits for a new message in the next iteration of the loop.

Processes can execute this loop at an arbitrary speed. Furthermore, the speed of a single process can change during its lifetime as well. Therefore, no assumption about the time required to complete an action can be made.

Processes also have an *internal state*, which consists of a collection of values stored and which is accessible by the process across its actions. The memory of a process is assumed to be unbounded. The internal state of a process p can be only directly observed or modified by p itself. Only if the interface of p permits it, other processes might access p's state by sending messages to it.

2.2 Interprocess Communication

Processes communicate with each other by sending messages over an asynchronous network. The network consists of a set of point-to-point connections called *links*. Two processes can communicate with each other if there exists a link connecting the two. A link is not equal to a physical connection between two machines. Depending on the underlying network topology, it might be necessary for a message to traverse multiple physical machines before reaching its destination (or none at all if both processes exist on the same machine). For the sake of simplicity, links are considered to be bi-directional. Furthermore, it is assumed that all pairs of processes are connected via links. This means any process can send messages to all other processes in the system.

Links provide *send* and *receive* actions that processes can use to send or receive messages, respectively. Process p sending message m to q is denoted as $send(m)_{p,q}$. Analogously, $receive(m)_{q,p}$ means that p receives m sent by q. If the sender and receiver of m is clear by context, only $send(m)$ or $receive(m)$ might be used. Since the network is asynchronous, no fixed communication rounds exist. Processes can use *send* and *receive* actions at any time, if they choose to do so.

Similar to processes, links operate at arbitrary speed. Messages can be delayed indefinitely before being delivered to their destination. It is important to note that a message arriving at a process does not equate to the process calling *receive*. Messages delivered by a link, but not yet fetched by the process are stored in a FIFO (first in, first out) message buffer with unbounded memory. As soon as a message is stored in a process's message buffer, the process can retrieve it by using *receive*.

Send actions are non-blocking, which means a process sending a message immediately continues its next instruction and does not wait until the message has been delivered or received. A *receive* action immediately returns the oldest

delivered message to the calling process. If no such message exists, *receive* will block the process and wait for it.

To simplify the description of the algorithms in the later parts of this thesis, it is assumed that the receiving process of a message always knows the sender of the message. This can be achieved simply by adding the sender's PID to all messages. Often, processes communicate with each other via a request-response scheme. If a process receives a response, it is assumed to always know for which request this is the reply for. This applies independently to the number of requests the process sent previously.

2.3 Messages

In the context of this thesis, messages are considered to be tuples, i.e., finite ordered lists of elements. Each element in a message can be of arbitrary structure and size.

By convention, the first element of a message is reserved for its *type*. Messages of the same type have the same number of elements, with each element representing the same kind of information. A message of type t consisting of elements $e_0, e_1, .., e_n$ is denoted as $\langle t, e_1, .., e_n \rangle$ (with $t = e_0$).

2.4 Failures

Many distributed systems used in practice involve a large number of components interacting with each other. In such a setting, failures become unavoidable. Amazon's Dynamo paper [11] describes the situation fittingly:

> "Dealing with failures in an infrastructure comprised of millions of components is our standard mode of operation; there are always a small but significant number of server and network components that are failing at any given time."

The number and types of failures a distributed algorithm must be able to handle has a big influence on its design. This section aims to give a short (and incomplete) overview over some important classes of failures relevant to this thesis.

2.4.1 Process Failures

Every process in a distributed system is supposed to faithfully execute the actions assigned to it. In general, a process which processes an infinite number of messages in such manner is called *correct*. It might also diverge from its expected behavior. In this case, the process is *faulty*. Depending on the environment a process lives in, different kind of failures must be considered.

Various criteria to classify failures can be found in literature. Typically, classes of failures are grouped into hierarchically ordered failure models. A failure model in such a hierarchy covers at least all failures included in any hierarchically lower level. Failure models commonly found in literature [19, pp. 29-34][12][1] are: crash-stop, omission, crash-recover, and arbitrary. Although different or more fine-granular classifications exist (e.g. in [6, pp. 24-30] and [2]), only these four will be briefly discussed in this section.

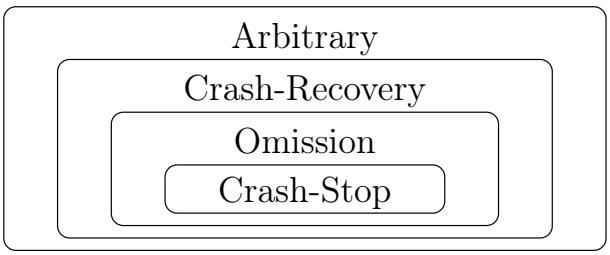


Figure 2.1: Hierarchy of process failure models (based on [19, p. 30]).

Crash-Stop This failure model covers *crash* failures. When a process crashes, it does not receive or send any messages and also stops executing any local instructions. Once crashed, it never recovers. In this model, a process is correct if it never crashes.

Omission A slightly more general kind of failure is *omission*. An omission occurs when a process deviates from its behavior by not receiving or sending a message when it was supposed to do so. Crashes are a sequence of omission failures where a process stops receiving and sending *any* messages after a certain time.

Crash-Recovery In some environments, it is possible for a crashed process to recover and to continue operating normally. All messages send to the process in the mean time are lost and it stops all local computations during its crash. The process might forget its internal state after its recovery. In fact, it is possible for the process to recover with any internal state it held at one point in the past. A process is called correct in this model as long as it crashed only a finite number of times and always recovers in a finite amount of time. However, when reasoning about a specific time or time interval, e.g. during the execution of an algorithm, a crashed process is considered faulty if it does not recover during this time.

Arbitrary As implied by its name, the arbitrary failure model is the most general one. It covers all deviations of a process from its expected behaviour. No assumptions about the behaviour of a faulty process in this model can be made. These kind of failures are also called *Byzantine*, or *malicious*, because processes might actively try to sabotage the whole system. Tolerating Byzantine failures is typically very costly, but is the only viable option in scenario in which there is a risk that some processes are controlled by a malicious users.

The later parts of this thesis focus mostly on the crash-recovery model. While omission and crash faults are covered by this model as well, Byzantine faults will not be taken into consideration.

2.4.2 Link Failures

In an ideal environment, all messages sent over a link are eventually delivered to a process. However, this is not always a realistic assumption. Unreliable networking hardware might lose or duplicate some messages. A severe failure might even

cause the network to split into multiple parts which cannot communicate with each other. This is known as a *network partition*.

In order to argue about the behavior of an algorithm in a distributed system, the message delivery guarantees of the links used must be known. Link abstractions are used to define properties of links independent of the underlying hardware. The *fair-loss link* and *reliable link* abstraction are introduced briefly in this section. Both are commonly used in literature and are covered in more detail at [19, pp. 35-43]. For the course of this section, it is assumed that both the sending and receiving process do not crash and that the network is not partitioned.

Fair-Loss Links Essentially, a fair-loss link guarantees that any message sent is delivered with a non-zero probability. Messages are not systematically dropped. More precisely, the *fair-loss* property of such a link states, that if message m is sent infinitely often, then m is delivered an infinite number of times. Furthermore, a message sent a finite number of times is not delivered infinitely often (*finite duplication*) and any message delivered by the link was previously sent by some process (*no creation*).

Reliable Links A reliable link, sometimes also called perfect link, is the link abstraction most commonly assumed (often implicitly) in research literature [32, p. 459]. Its *reliable delivery* property guarantees that any message m sent will be eventually delivered, and m will be delivered no more than once (*no duplication*). As with fair-loss links, any message delivered was previously sent by some process (*no creation*).

Both link abstractions do not specify the order in which sent messages are delivered. If messages can arrive in any order, then the link is called *unordered*. A link delivering messages in the order they were sent is an *ordered* link.

Examples of implementations of both reliable and fair-loss links can be found in the internet protocol suite. TCP follows the idea of ordered reliable links, whereas UDP can be roughly seen as an implementation of unordered fair-loss links.

The applicable link abstractions usable for a distributed algorithm is not always obvious. As described later, Scalaris is using TCP for the communication between Scalaris nodes. Nevertheless, reliable links cannot be assumed in later parts of this thesis. The reason for that lies within Scalaris' underlying architecture and will be described in Section 6.1.

3 The Consensus Problem

The consensus problem is one of the most fundamental problems in distributed computing. It requires a number of processes to agree on a single value or action to take. An algorithm solving this problem is called a *consensus protocol*[1].

This work focuses on the *asynchronous* consensus problem, a variation in which processes can only communicate with each other by sending asynchronous messages (see Section 2.2).

[1] The distinction between a distributed algorithm and a protocol in literature is vague. Both terms are used interchangeably in this thesis.

3.1 Problem Statement

The traditional consensus problem is often described as a single set of processes reaching some kind of agreement [15][16]. Leslie Lamport, however, argues in [25] that the problem is better described in terms of a set of *proposer* and *learner* processes. The proposers can propose values independent from each other and the learners must agree on a single proposed value. Note that the set of learner and proposer processes does not have to be disjunct. Lamport states three safety requirements that a solution to the consensus problem must fulfill:

Nontriviality Any value learned must have been proposed.

Stability A learner can learn at most one value.

Consistency Two different learners cannot learn different values.

These requirements must hold under certain failure assumptions. In the context of this thesis, this means that any number of failures under the crash-recovery model must be tolerated. However, a process which has recovered from a crash is not allowed to forget its internal state.

In addition to the safety requirements, fulfilling the following liveness requirement guarantees that eventually an agreement is reached:

Liveness(v, l) If value v has been proposed, then eventually learner l will learn some value.

In contrast to the safety requirements, liveness is parametrized by learner l and value v. This is because it must hold under the assumption that the proposer of v, learner l and a sufficient number of other processes are correct. The exact number of correct processes required depends on the algorithm in use. For example, the basic version of the Paxos protocol (covered in Chapter 4) requires a majority of so-called *acceptor* processes to be correct.

Simply put, fulfilling the safety requirements ensures that nothing incorrect ever happens despite any number of failures, whereas fulfilling liveness ensures that eventually something correct happens if not too many failures occur.

3.2 Impossibility of Asynchronous Consensus

In spite of its fundamental nature, the asynchronous consensus problem is not an easy problem to solve. In fact, it was proven by Fischer, Lynch, and Paterson in [16] that there cannot exist a consensus protocol in a fully asynchronous setting that always reaches a consensus in finite time.

This result was achieved using a very weak form of the consensus problem. Processes communicate with each other using unordered reliable links and can fail according to the crash-stop model. Furthermore, only *some* of the correct processes are required to agree on the value 0 or 1. Despite these simplifications, they could show that even with only one faulty process, there exists for any arbitrary consensus protocol a message delivery order for which the protocol fails to reach an agreement.

This result, also known as the "FLP result"[2], settled a year-long ongoing dispute in the distributed systems research community. Solutions for the synchronous consensus problem already existed to this time, even when considering Byzantine failures, but it was unknown if a solution in the asynchronous case was possible [16].

The critical difference between the two variations is that process failures can be reliably detected in the synchronous case. Any process that does not reply within a certain time frame can be assumed to have crashed. But no upper limit for the response time of a process exists in a fully asynchronous setting. Therefore, a process that is just very slow (or is connected via very slow links) is indistinguishable from a crashed process [9].

Due to the FLP result, no consensus protocol can fully satisfy the liveness requirement described in the previous section without further assumptions. Some approaches require the correctness of certain processes or assume an upper bound for communication delays. Other protocols are designed so that any undecided state is unstable. As time goes on, the probability of reaching an agreement approaches 1 asymptotically. One such protocol is Paxos. It will be described in the following chapter.

4 Paxos

This chapter introduces the Paxos algorithm discovered by Leslie Lamport. It solves the asynchronous consensus problem described in the previous chapter. Lamport introduced his algorithm first 1998 in [29], using a fictional parliament on the Greek island Paxos as an allegory. He later realized that this was a mistake, as many were so distracted by the setting that the significance of the algorithm was lost [28]. Lamport then republished his idea 2001 in [26].

Today, a number of Paxos variations exist which optimize various properties of the original proposal. Examples include the number of message delays needed to reach a consensus (Fast Paxos [24]), the types of failures tolerated (Byzantine Paxos [23]) and the number of failures tolerated (Cheap Paxos [31]). Protocols of the Paxos family are used internally in many different distributed systems. These include the lock service Chubby [5], the database Spanner [10], the file system XtreemFS [22] and the key-value store Scalaris [38].

The following sections focus on the basic, unmodified version of Paxos, as it is described in [26]. Most of the concepts introduced here are later reused in the Paxos variation used in Scalaris, PRBR (Paxos Round Based Register), which will be described in Section 7 of this thesis.

4.1 Process Roles

Paxos uses three roles to describe the interaction between processes: *proposers*, *learners* and *acceptors*. Every process in the system may perform one or multiple roles. The actual mapping from processes to roles is implementation dependent and has no effect on the correctness of the protocol.

[2] Named after its discoverers: Fischer, Lynch, and Paterson

In a typical setting, a client submits a request to the distributed system by sending a message to one of the proposer processes. The proposer receiving the request then tries to convince a sufficient number of acceptors to agree on the proposed value. Acceptors are sometimes referred to as voters. Each acceptor keeps track of its own vote and can deny or accept any new arriving proposals. At any given time, an acceptor has voted for at most one value, but it can change its vote if a new proposal arrives. Acceptors act independent from each other and might accept different proposals at the same time. Once a sufficient number of acceptors agree on the same value, it is possible for a learner process to learn this value and notify the client of its result.

4.2 Quorums

To satisfy the $Liveness(v, l)$ requirement of consensus, a sufficient number of acceptors, in addition to the proposer of v and learner l, must be correct during the execution of the protocol. Any set of acceptors that is large enough is called a *quorum*. If less than a quorum of correct acceptor processes remain, then the protocol can not learn any value. At the same time, quorums maintain the safety requirements by ensuring that at least some acceptors retain knowledge of any previous result. Therefore, any two quorums must share at least one acceptor. Let $Quorum$ be the set of all quorums.

Definition 1 (Quorum)**.** *A quorum is a set of acceptors large enough to learn a value. The* ***Quorum Property*** *states that any two quorums must have a non empty intersection:*

$$Q_1 \cap Q_2 \neq \varnothing, \quad \forall Q_1, Q_2 \in Quorum \tag{4.1}$$

To understand why this is important without further knowledge of the algorithm, assume a case in which there exist two quorums Q_1 and Q_2 with an empty intersection. Now, two concurrent proposals, made by different proposers, might each convince quorum Q_1 and Q_2 respectively to vote for their proposal. In fact, both proposals do not know the existence of the other because there exists no process involved in both proposals. Since both proposals happen at the same time, two learners might learn different values. This violates the consistency requirement of consensus.

In the context of basic Paxos, the quorum property is typically satisfied by defining a quorum to be any *majority* of acceptors. This definition will be also adopted in this thesis. Let N be the number of acceptors in the system.

$$|Q| \geq \lfloor N/2 \rfloor + 1, \quad \forall Q \in Quorum \tag{4.2}$$

Quorums can be chosen to be larger than this without changing the correctness of the protocol. Although this will reduce the protocol's performance since the quorum size has a direct influence on the minimal number of messages sent before a decision can be made.

More importantly, the size of quorums impacts the number of faulty processes that can be tolerated. If the system is required to survive F failures, at least $2F + 1$ acceptors must be used.

When reasoning about the behavior of Paxos, it is sometimes useful to reason about sets of acceptors which are not large enough to constitute a quorum. Such a set is called a *minority*.

Definition 2 (Minority)**.** *A minority of acceptors is a set of acceptors which is not a quorum.*

This definition of minority has a slightly different meaning than its conventional usage, i.e., forming less than half of something. Depending on the minimum size of quorums used, a minority of acceptors can also contain half or more than half of all acceptors.

4.3 The Algorithm

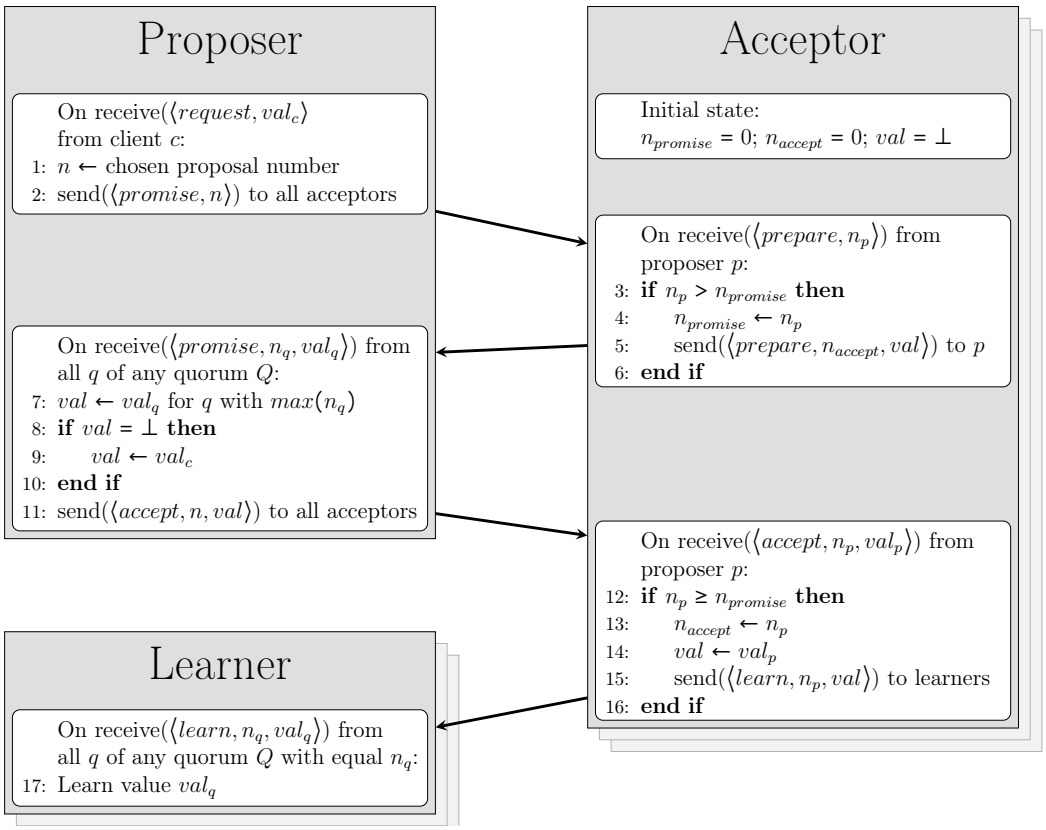
The basic Paxos algorithm proceeds over several rounds. A successful round has two phases. The first phase of a round starts with a client c sending a proposal with an arbitrary value val_c to a proposer p. Code Listing 4.1 depicts the following algorithm and can be used as a reference.

Any proposer might receive multiple proposals at the same time. Therefore, p first chooses a proposal number n, which should be greater than any number chosen previously by p. Proposal numbers are used to distinguish concurrent proposals during the execution of the algorithm. They are required to be unique, but they are not required to be natural numbers as long as they can be totally ordered. A possible way to ensure unique proposal numbers across proposers is to express them as a 2-tuple $\{i, pid\}$, where i is a natural number and pid the respective proposer's process ID. Since process IDs are unique, all proposal number are unique as long as a process does not choose the same natural number twice. If proposal number n is selected, then this execution of the algorithm is referred to as *round n*.

Once p has chosen n, it sends a *prepare* message containing n to all acceptors (line 1-2). All acceptors keep track of the highest proposal number received in *prepare* messages, as well as the latest value with its corresponding proposal number the acceptor has voted for. Any acceptor a receiving p's message checks if it already received a different *prepare* message with a higher proposal number (line 3). If this is the case, a will ignore the message[3]. Otherwise, a replies to p by sending a *promise*, which includes the value a currently voted for, as well as the round in which a voted for this value. By sending this promise, a effectively agrees to not accept any proposals executed in a lower round than n (line 4-5). This concludes the first phase.

Once p has received promises from any quorum of acceptors, it starts the second phase of the algorithm. In this phase, p chooses a value which will be hopefully learned by the learners. If any of the received *promise* messages contain a value an acceptor has previously voted for, then p chooses the value of the highest received round. Otherwise, p can choose its own value val_c (line 7-10). p now sends an *accept* message with the chosen value and n back to all acceptors. Note that it is implicitly assumed that p remembers both n and val_c across its actions.

[3]For the sake of optimization, a can notify p that a higher numbered proposal exists so that p can abandon the current proposal and retry with a higher number. However, this is not part of the algorithm.



Code Listing 4.1: Pseudocode of the basic Paxos algorithm.

Any acceptor a receiving the *accept* message will again check if it has given a promise for a higher numbered proposal (line 12). In this case, a ignores the message. Otherwise, a will change its vote to the received value and updates its respective proposal number accordingly (line 13-15). Afterwards, a notifies all learners of its new vote.

If at any point in time, a quorum of acceptors has voted for the same value in the same round, then this value will be learned eventually (under the assumption that the liveness requirement is fulfilled). Such a proposal is called *chosen*. A learner learns this value if it has received a message from a quorum of acceptors in the same round (line 17). This concludes the algorithm.

Notifying each learner whenever an acceptor has changed its vote requires a high number of messages. The message count to learn a value is at least the product of the number of acceptors and learners. An alternative approach uses a distinguished learner, which will be the only learner receiving messages from acceptors. Once this learner learned a value it notifies the other learners in the system. This approach requires an extra message delay. It is also less reliable since the distinguished learner could fail. However, this approach requires only the sum of acceptor and learner processes of messages.

What it means for learner l to learn a value depends on the context the algorithm is deployed in. In some case, l simply notifies the client c of the chosen value. In a more elaborate setting, the set of learners might act as a distributed storage. The learned value is persisted and returned upon request. This is especially useful when Paxos is employed to learn a *sequence* of values, for example when implementing

a replicated state machine. Both sequences of consensus and replicated state machines will be further discussed in Chapter 5.

4.4 Correctness

The structure of Paxos is simple. However, it is not obvious to see how the algorithm satisfies the safety requirements of consensus. The section aims to give some insights about the correctness of Paxos. It is not intended as a proof of correctness. A more rigorous line of reasoning can be found in [26] or [29].

Nontriviality is easy to see. Before any learner can learn a value v, a quorum of acceptors must agree on v in any round. Since an acceptor will only vote for v if it was proposed by a proposer and Byzantine faults (which may alter the proposed value) are not covered, nontriviality will not be violated.

Once an acceptor voted for a proposal, it will never vote for a lower numbered proposal. Due to the quorum property, once a proposal numbered n with value v is chosen, it is impossible for a lower numbered proposal to be chosen. This is because only a minority of acceptors which are voting for a lower proposal than n remain. To satisfy consistency and stability it therefore suffices to ensure that any chosen proposal with a higher number than n will also contain v.

Any proposer starting a new round after the proposal containing v was chosen will receive at least one reply containing v in its promise messages.

It is also not possible that any acceptor has voted for a higher numbered proposal than n with a different value than v before proposal n was chosen. This would imply that a proposer received promises with a proposal number higher than n from a quorum of acceptors that have not already accepted the proposal with v. But this leads to a contradiction because there would not exist a quorum of acceptors anymore which could accept proposal numbered n.

Therefore, at the time proposal n was chosen, no acceptor voted for a value of a higher numbered proposal with a different value. This means the highest accepted value returned in promise messages for all new rounds will be v. This causes all proposers to choose v in any subsequent higher numbered proposals. Since no value other than v will be proposed, no learner will ever learn a value different from v.

4.5 Liveness

The liveness requirement operates under the assumption that a proposer, learner and a quorum of acceptors are correct. However, liveness cannot be guaranteed without further considerations and assumptions.

The first problem which must be considered is message delivery. Paxos can be used using fair-loss links. This means a message can be lost with some non-zero probability. If too many messages are lost, a proposer or learner might not receive a sufficient number of replies from acceptors to continue with the algorithm. In this case, no further progress can be made (unless some proposer starts a new round of the algorithm).

Since communication is asynchronous, the proposer has no way of knowing whether the message is delivered very slow or if it is lost completely. This problem can be solved when assuming an upper bound for communication delays by having

the proposer to retry with a new Paxos round after a certain amount of time. Due to delivery guarantees of fair-loss links, there will be eventually a round in which enough messages will be delivered. When perfect links are used, this is not necessary since all messages will be delivered eventually.

The second problem is known as *dueling proposers*. Concurrent proposals of two or more proposers can block each other indefinitely in the following scenario:

Proposer p_1 completes phase 1 with a proposal number n_1. A second proposer p_2 also completes phase 1 with a higher proposal number n_2. p_1's accept messages will now be denied or ignored in phase 2. Therefore, p_1 begins in phase 1 again in a round with a higher proposal number, thus blocking p_2 from succeeding. In this scenario, no value is ever learned.

Fortunately, this state is unstable. If proposers, acceptors, or links operate at slightly varying speeds, then one proposal will be eventually chosen. The probability of such a sequence of denied proposals can be further reduced by introducing randomness for retrying a proposal. Proposers can simply wait for slightly different intervals before starting a new round or use techniques such as exponential back-off. However, this is not enough to *guarantee* liveness. Especially in settings with many concurrent proposals, the probability to never recover from dueling proposers is non-zero.

A solution to this problem is introduced by Lamport in [26, p. 7]. A distinguished proposer called *leader* must be selected, which is the only proposer allowed to issue proposals. Other proposers must communicate with the leader which in turn coordinates all incoming proposals.

This approach has some drawbacks. First, the leader becomes a bottleneck. A slow leader might slow down the system if too many proposals arrive in a short period of time. Second, the leader is a single point of failure. Once the leader crashes, a new leader must be elected. However, leader election is a specific kind of consensus problem and is thus subjected to the FLP result [13]. If the leader election fails and more than one leader is elected, the protocol can still operate normally, but dueling proposers are still possible in this case.

In summary, guaranteeing liveness requires that a single leader, a quorum of acceptors, and a learner do not fail. In addition, when using fair-loss links an upper bound for communication must be assumed.

5 Consensus Sequences

The previous chapters focused on a single instance of the consensus problem. That is, achieving agreement of multiple processes on a single value. While this is useful, many scenarios require it to establish a sequence of values – one of the most prominent examples being *replicated state machines*.

The replicated state machine approach was first proposed by Lamport 1984 in [30] and was further elaborated in 1990 by Fred Schneider in [36]. Examples of current systems making use of replicated state machines include ZooKeeper [21] and Chubby [5] and Scalaris [38].

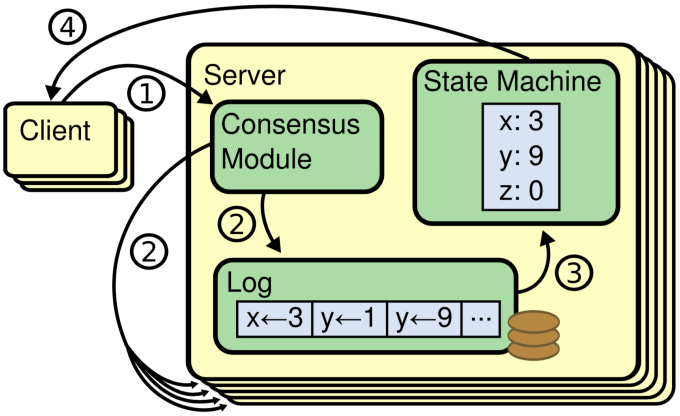


Figure 5.1: Replicated state machine architecture [35].

5.1 Replicated State Machines

A state machine, also known as automaton, is a computational model. It consists of a set of state variables and transition functions. The state variables describe the current state of the state machine, whereas transition functions manipulate the state variables, thus causing the state machine to transition into another state [32, pp. 200-204].

Processes, as they are described in Section 2.1, are also state machines. The collection of variables held by the process describes the state of an analogous state machine. The process's actions can be mapped to transition functions.

In the context of distributed systems, servers can provide services by implementing state machines. Clients can interact with servers by issuing *commands*. Commands are executed by a server by applying the respective transition function of its state machine. A command can change the server's state (writes), produce output (reads), or both. Commands are executed atomically with respect to other commands. They are either applied in full or not at all [36].

When providing a reliable service it is important to ensure that the failure of a single server has no or only minor impact on the service quality experienced by clients. This can be achieved by introducing redundancy to the system. Instead of having only one instance of the state machine on one server, identical state machines are deployed on multiple servers. This is called a replicated state machine. A single instance of a state machine in a replicated state machine is called a *replica*. If two replicas on two servers are in the same state, executing the same command on both servers will lead to the same result. Naturally, this requires all commands, and by extension the state machine, to be deterministic (i.e. there is no randomness involved in the execution of a command).

Figure 5.1 depicts the typical architecture of a replicated state machine [35]. Clients issue requests by sending a command to the consensus module of one of the servers implementing the replicated state machine (step 1). The consensus modules of all server instances communicate with each other and are implementing an arbitrary consensus protocol, e.g., basic Paxos. Each server typically stores the learned commands in a replicated log (step 2). Due to the properties of consensus protocols, all logs contain the same commands in the same order. Once a command

is included in a log, the respective state machine can execute it (step 3). If required, the command log can be persisted on a local stable storage. This allows the server to recover with a previously known state. If it missed some commands in the mean time, it can fetch them from the other instances [8].

An explicit command log is not necessarily required. Some systems, including Scalaris, do not keep such a log. Instead, they apply any learned command directly as soon as it is learned.

Since state machines and commands act deterministic, each replica will traverse the same states and produce the same sequence of outputs. Any server can therefore return the result to the client who issued the respective command (step 4). From a clients perspective, the whole system acts as a single entity.

A replicated state machine can tolerate as many failures as the consensus protocol in use. When using Paxos, each server typical contains a learner, proposer, and acceptor process. Therefore it can tolerate at most F failures if $2F + 1$ replicated servers are used. Due to Paxos' properties, a specific command will be learned as soon as a quorum of acceptors (i.e. a majority of replicas) has agreed on it. Therefore, any instance in this quorum can return the result of the command before the remaining minority has learned it.

5.2 Requirements for Consensus Sequences

Section 3.1 defines the requirements of a consensus protocol for learning a single value. These must be generalized when reasoning about learning an increasing sequence of values. Because all further chapters of this thesis focus on consensus sequences in replicated state machine, the term *command* instead of the more generic term *value* will be used henceforth.

In the conventional state machine approach, some learners might learn commands out of order. This might happen because of lost, or out of order delivered messages. A learner might learn the 5^{th} command before learning the 3^{rd} command. But it knows that the respective command is the 5^{th} one. However, it cannot execute the 5^{th} command immediately, because all replicas must apply all commands in the same order. Therefore, it is convenient to define that a learner can only learn a command if it has already learned all previous commands. This way all learners always learn a gapless, ever increasing sequence of commands [25].

The safety requirements can now be generalized as follows [25]:

Nontriviality Any learner has always learned a sequence of proposed commands.

Stability The sequence of commands a learner has learned at any time is a prefix of the learned sequence at any later time.

Consistency For any two learners, it is always the case that one of the learned sequences is the prefix of the other.

Note that the prefix relation is reflexive: Any sequence is the prefix of itself. The safety requirements must hold under the same assumptions as discussed 3.1. To reiterate, any number under the crash-recovery must be tolerated, with the additional restriction that no recovered process forgets its internal state.

Analogous, the liveness requirement can be generalized as:

Liveness(c, l) If command c has been proposed, then learner l's learned sequence will eventually contain c.

Again, to satisfy liveness in Paxos, at least the proposer of c, leaner l and a quorum of acceptors must be correct during the execution of the protocol until c is learned.

In contrast to the original liveness requirement, where *any* proposed command must be learned, *all* proposed commands must be learned now. Although the order in which commands can be learned is arbitrary, as long as the order is consistent across all learners.

5.3 Strong Consistency

A consensus protocol that satisfies the safety requirements stated above allows it to implement a replicated state machine which adheres to the strong consistency model. Strong consistency, also known as linearizability, was first defined by Herlihy and Wing in [20] and is often regarded as the "ideal" correctness condition a distribution storage should aim for [41].

Roughly speaking, strong consistency requires that each operation must appear to be applied instantaneously at some point in time between its invocation and response [41]. This point in time is also known as linearization point, whose existence was first postulated by Lamport in [27]. The precise location of the linearization point during the execution of an operation is not known from the outside. Is a write and read operation executed concurrently, the read operation may return the state without application of the write. However, once any read has observed the new state, all subsequent reads are not allowed to return the old state. This also includes reads submitted by different clients.

Figure 5.2 gives an example of strong consistency by using a register holding a single value. The register is initialized to 0. P_C's read is concurrent to both of P_A's writes. The read is allowed to return either 0 or 1. It can not return 2 since a subsequent read executed by P_B has not seen the application of the second write. Would P_B's read return 2 or would it be executed concurrently to P_C's read, then it would be possible for P_C's read to return 2 as well.

The concept of linearization points can be easily applied to many Paxos variants. In the basic algorithm, as described in Section 4.3, acceptors could vote for a number of values in ascending rounds. Once a quorum of acceptors voted for the same value in the same round, this value was considered to be chosen. This is the exact point in time at which the consensus decision was made since the chosen value will be eventually delivered and no value is delivered before a chosen value exists. As will be seen throughout this thesis, extending Paxos to a consensus sequence usually entails learning a sequence of chosen commands. The moment

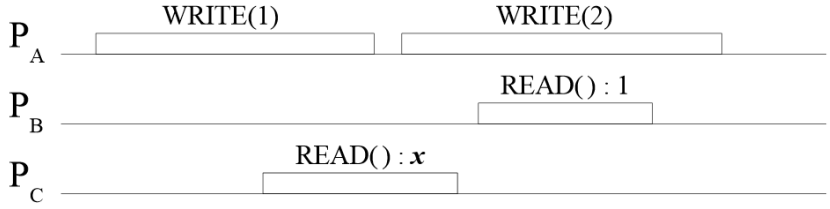


Figure 5.2: An execution exhibiting read-write concurrency [41].

such a command is chosen coincides with the linearization point of the respective commands under the strong consistency model.

5.4 Paxos in Replicated State Machines

The basic Paxos algorithm can only be used to get agreement on a single command. Lamport originally described in [26] how Paxos can be used as a building block to achieve consensus on a sequence. This extension is commonly referred to as Multi-Paxos [8].

In Multi-Paxos, a new instance of basic Paxos is used whenever a new command needs to be learned. Each Paxos instance is assigned to a slot. If a server wants to learn the i^{th} command, it will use the Paxos instance in slot i. Since Paxos satisfies the safety conditions of consensus, at most one command can be learned for every slot. As noted in Section 5.2, a command is only considered to be learned if all previous commands have been learned before. By this definition of learned, a command can be executed immediately as soon as it is learned.

It is easy to see that Multi-Paxos fulfills safety for command sequences, since each instance of Paxos fulfills these requirements for choosing a single command. As soon as a slot is filled, the learned command will never change. Because previously learned commands never change and only the next empty slot can be learned next, stability is ensured. An analogous argument can be applied for consistency and nontriviality.

If a proposer proposes a value for a specific slot, its proposals might fail due to a concurrent proposal with a higher round number. In this case, the proposer can simply retry its proposal for a higher slot. Under consideration of the additional assumptions made for the basic Paxos algorithm in Section 4.5, the proposer will eventually have success with its proposal, thus satisfying liveness.

The setup described above requires that every Paxos instance completes both phases of the algorithm. Therefore, at least four message delays are needed (or more if there are dueling proposers). This can be improved by electing a distinguished leader as coordinator across multiple Paxos instances. When a new leader is elected, it executes phase 1 of the algorithm. But instead of executing it for a single Paxos instance, this exchange is valid for infinitely many instances following the current instance. As long as the leader does not change, the first phase can be skipped now for all following instances by using the same proposal number every time. If another server wants to be the leader (e.g. because the previous leader failed), it can simply execute phase 1 of the protocol himself with a higher proposal number. Due to this optimization, only two message delays are needed to learn a command as long as the leader does not fail. However, the command must be first transmitted to the leader, which adds a message delay. In addition, the same drawbacks as discussed in Section 4.5 in respect to a distinguished leader apply here as well.

Solving the problem of a consensus sequence by chaining multiple instances of Paxos sounds simple in theory. However, a number of non-trivial challenges arise when trying to implement Multi-Paxos – most notably in the management of Paxos instances. Every new instance of Paxos created takes up some amount of resources. Therefore, it is important to eventually remove old instances for commands that have been learned by all replicas. It is, however, not obvious when

any given instance can be removed safely. It is not safe for any given replica to remove a Paxos instance just because the learner process for this instance has learned the command, since it is possible that the learners of the other replicas were unable to learn it, e.g., due to message loss. Removing the state of one of the acceptors might cause a quorum of acceptors to exist which does not know the existence of the proposed command. Furthermore, once a quorum of replicas has freed their instance, it is not clear how the remaining minority can reliably learn that their instance can be removed as well.

The management of separate Paxos instances adds an additional level of complexity and subtlety to the system. However, instance management is a topic rarely discussed in literature. Lamport's original description of Multi-Paxos [26] only sketches a possible approach and no widely agreed-upon algorithm of Multi-Paxos exists. The creators of Raft (a different consensus algorithm) argue that these are some of the main reasons that Paxos is widely considered to be hard to understand and implement [35].

The algorithm used in Scalaris uses a different approach than Multi-Paxos. Instead of using multiple Paxos instances for different slots, PRBR (Paxos Round Based Register) tries to establish a sequence of commands using only a single Paxos instance. This way, no Paxos instances have to be dynamically created and destroyed, thus preventing the problems and overhead caused by Paxos instance management. PRBR's general idea is it to only propose a new command (i.e. start phase 2 of the basic algorithm) if the proposer is certain that the previous command has been learned already. How this is achieved will be discussed in more detail in Chapter 7.

5.5 Commutative Commands

Conventional implementations of the consensus module in replicated state machines bring all arriving commands in a strictly sequential order. However, this is not always necessary.

Usually, state variables, henceforth referred to as *values*, held in a replicated state machine are not dependent on all other values. In some instances, values are independent to all other values stored. This can be the case if, for example, the replicated state machine is used as a foundation for implementing a distributed key-value store. As opposed to relational databases, a key-value store does not maintain relationships between values. Assume a scenario in which two clients concurrently submit commands c_1 and c_2, respectively. The commands access different values. If both values are independent of each other, it does not matter if c_1 or c_2 is executed first. In both cases, the end result for the client and the state machine is the same. By extension, individual instances of the replicated state machine can execute the commands in either order. During the execution of the commands, the internal states might not be the same for all instances. But after all instances have executed both commands their internal states match again. Since the replicated state machine acts as a single entity in respect to the clients, they do not notice any inconsistencies.

In the scenario described above, c_1 and c_2 are said to *commute* with each other. In converse, two commands *interfere* with each other if they must be applied in the same order by all replicated state machine instances . A more rigorous definition of

commuting and interfering commands is given in Section 8.1. A common example for commutative commands is two concurrent reads. A read does not change the internal state of a state machine instance. Therefore, any two reads are commuting.

The rate of interfering commands is often low for practical systems. For example, over 90% of requests handled by Chubby deal with lease renewal traffic [33]. Leases grant clients exclusive access to resources over a restricted time frame. The client must periodically renew the lease if it does not want to lose access. Because only the client owning a lease can renew it, two concurrent lease renewals by different clients must target different leases. Therefore all lease renewal commands commute with each other.

Strictly ordering commutative commands in a sequence causes coordination overhead. This effect is especially pronounced when using Paxos without a coordinated leader, since two concurrent requests can cause dueling proposers. Chapter 8 of this thesis focuses on taking advantage of the commutative relationships of concurrent commands in PRBR by not requiring a strict sequential execution order for all commands of a replicated state machine.

6 Scalaris

Scalaris [38] is a distributed key-value store featuring ACID (Atomicity, Consistency, Isolation, Durability) compliant multi-key transactions. While traditional SQL based relational databases always provide such transactions, many comparable distributed databases have to relax the ACID properties in order to accomplish full decentralization.

The CAP theorem [18] states that any distributed system can only achieve at most two of the following properties: strong consistency (C), availability (A) and partition tolerance (P). Scalaris provides strong consistency and partition tolerance at the expense of availability. This is achieved by using a replicated state machine approach with a modified version of the Paxos consensus protocol (PRBR) on top of a structured overlay. In case of a network partition, the partition with a quorum of replicas (and thus acceptors) can still provide consistent transactions. However, the minority partition cannot answer any requests until the partitions are reconnected. Therefore, the availability property of the CAP theorem is not fulfilled. The focus on strong consistency and multi-key transactions sets Scalaris apart from other distributed key-value stores like Amazon's Dynamo [11], which only implement weaker consistency properties in favor of better availability.

Scalaris is an ongoing research project at Zuse Institute Berlin [3] and is written in Erlang[4]. It is fully open-source and is published under the Apache Licence 2.0. The source code is available on GitHub[5].

6.1 Architectural Overview

In a typical setup, Scalaris is deployed across multiple machines. An instance of Scalaris running on a machine is called a node. Each node consists of a number

[4] `http://www.erlang.org/`

[5] `https://github.com/scalaris-team/scalaris`

of processes as described in Section 2.1. The total number of Scalaris nodes used is denoted by N.

A node can be divided into four layers [4], which will be briefly described in this section.

Unstructured Peer-to-Peer Layer The bottom layer handles communication between processes of any two nodes. TCP is used as the transmission protocol, thus providing reliable links. Since Erlang's communication model is based on message passing, processes can communicate with each other in a similar manner as described in Section 2.2.

Structured Overlay Network This layer provides an implementation of a distributed hash table (DHT). A DHT allows storage and lookup of key-value pairs similar to a conventional hash table. When accessing an item in a hash table, a globally known hash function is used to map the item's key to the key space of the hash table. The hashed key represents the location of the item in the table and can be used to retrieve or modify it. For the sake of conciseness, the term *key* will be used for an item's key as well as its hashed counterpart.

Unlike hash tables, the key space of a DHT is split into multiple partitions of arbitrary size, which are then each assigned to a process. A process can only directly handle requests concerning some key k if it has ownership of the partition containing k. Otherwise, it must forward the request to another process by using a routing protocol. The number and size of partitions can change over time, e.g., due to processes joining or leaving the DHT.

The structured overlay network layer hides the management of the key space from all upper layers. It provides a generalized lookup mechanism for sending messages to nodes responsible for a given key. In the context of this thesis, this layer can be seen as the implementation of a link abstraction. Due to the routing of messages, a number of intermediate Scalaris nodes might have to examine a message before it reaches its destination. If a node crashes, all messages it is currently processing for routing purposes are lost. Therefore, the fair-loss link abstraction must be assumed for all layers on top of this layer.

The DHT implementation used in Scalaris by default is based on Chord [39] which models the key space as a ring structure and provides a routing mechanism with logarithmic worst time complexity. Although other DHT implementations such as Chord# [37] and Flexible Routing Tables [34] exist, all following explanations assume Chord as the DHT used.

Replication Layer To improve availability in case of a node failure, each value is replicated and stored in multiple locations (i.e. keys) on Chord's ring. For that, Scalaris uses a configurable *replication degree* R to store each item R times using R keys. The keys are calculated using a symmetric replication scheme [17]. To reduce the probability that the crash of a single node is affecting multiple keys of the same item, practical configurations use $N > R$.

Section 5.1 introduced the concept of replicated state machines as an architectural design to replicate a set of value across multiple, independent processes. In Scalaris, this approach is used on a *per-key basis*. Every stored item is managed in its own replicated state machine. A Scalaris node which is responsible for n keys

can therefore contain up to n replicas belonging to potentially n different state machines.

To ensure consistency across replicas, the aforementioned extension of the Paxos protocol, PRBR, is used. Arbitrary write and read commands can be submitted to read or modify the value of a single key. PRBR establishes a separate consensus sequence of commands for every key. Therefore, any two concurrent commands targeting different keys will not interfere with each other by default. In addition, PRBR can restrict valid follow-up commands based on the current state of the consensus sequence by using a so-called *ContentCheck*. This makes it possible to define conditional writes, which are useful when storing more complexly structured data in PRBR.

Transaction Layer Since PRBR can handle arbitrary commands, it can be used as basis for any replicated data type. Most commonly, PRBR is used as a basis for an optimistic transaction protocol providing ACID compliant transactions.

In contrast to pessimistic methods, an optimistic transaction protocol operates under the assumption that transactions can be completed without interference in the majority of cases. While running, transactions can read multiple keys and perform modifications on them locally. Before committing the resulting modifications to the system, it is verified that no conflict to other finished or validating transactions arose. During commit of the transaction's result, the accessed keys are briefly locked to ensure atomicity when modifying multiple keys in a single transaction.

The transaction protocol requires that each key maintains some meta information (read locks, write locks, version) in addition to the actual value. PRBR's interface allows it to submit read and write commands that only access or modify parts of a value. This allows the implementation of operations such as lock manipulation without unnecessary data transfer costs.

The transaction layer defines an interface which allows arbitrary applications to use Scalaris as a consistent, fault-tolerant backend which can scale at runtime by adding or removing Scalaris nodes if required.

Part II
Paxos Consensus Sequences

7 Paxos Round Based Register

7.1 Preliminaries

Paxos Round Based Register is an extension of the basic Paxos algorithm which allows establishing a consensus sequence of proposed commands. The algorithm aims to satisfy the safety requirements of such sequences as stated in Section 5.2 and aims to provide strong consistency. In contrast to Multi-Paxos, PRBR establishes sequences of consensus decisions in-place. This means only a single PRBR instance is needed, which eliminates the need for instance management and all related implementation challenges.

PRBR is still work in progress. At the current time, no publications related to PRBR exist. One goal of this thesis is it to provide a first description of the protocol, including a sketched argumentation about its correctness. The primary sources of information for this chapter are PRBR's implementation in Scalaris[6], along with a drafted, currently unpublished document about Scalaris' architecture and numerous discussions with Dr. Florian Schintke, one of the creators of PRBR.

7.1.1 Register of Consensus Sequences

The basic Paxos algorithm uses the roles of proposer, acceptor, and learner to describe the interaction between processes in the system: The proposers propose commands, a quorum of acceptors must agree on a command, and the learners learn the agreed upon command. The state machine approach in conjunction with Multi-Paxos described in Section 5.4 extended the single consensus to a consensus sequence by using multiple chained Paxos instances. But since only a single consensus sequence is used to manage a set of values, coordination overhead is necessary for concurrent commands even if they access independent values.

PRBR solves this problem by managing an independent consensus sequence for each key. This is achieved by splitting the set of acceptors into multiple disjunct subsets. Each subset is responsible for agreeing on commands for a given key.

Definition 3 (k-acceptor)**.** *A k-acceptor is an acceptor which can vote for proposals regarding the item with key k.*

When sending a message to a k-acceptor regarding a key other than k, it can ignore the message or respond with an appropriate deny message. A command proposed for key k can be learned as soon as a k-quorum of k-acceptors has agreed on it.

[6] https://github.com/scalaris-team/scalaris/tree/master/src/rbr

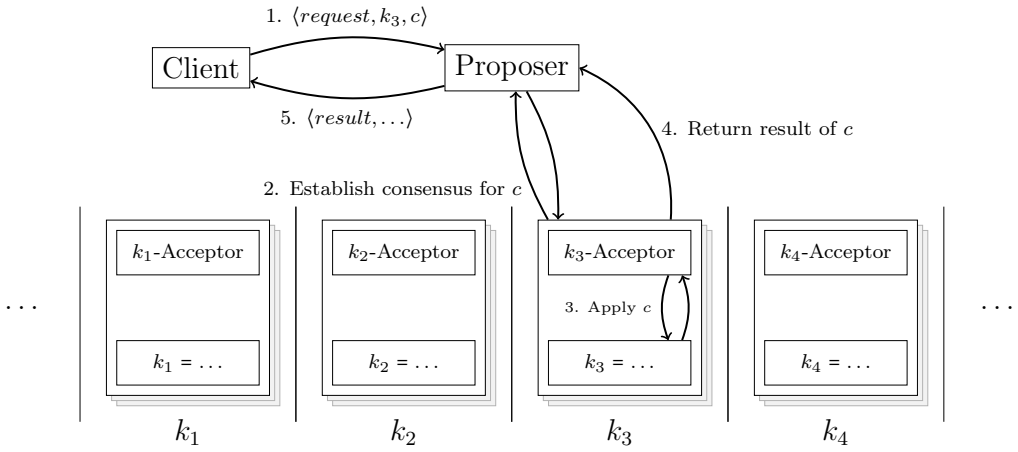


Figure 7.1: Proposing a command for key k_3 in PRBR.

Definition 4 (k-quorum)**.** *A k-quorum is a set of k-acceptors large enough to learn a proposal regarding the item with key k. The* ***k-Quorum Property*** *states that any two k-quorums must have a nonempty intersection.*

As with quorums of basic Paxos, the size of k-quorums can be freely chosen as long as the k-quorum property is fulfilled. In the context of this thesis, any proper majority of k-acceptors is considered to be a k-quorum. Since the set of k-acceptors is disjunct from the set of j-acceptors for all $k \neq j$, no k-quorum will intersect with a j-quorum. Therefore, two concurrent proposals for different keys cannot lead to dueling proposers anymore.

Analogously to k-acceptor and k-quorum, the term k-minority is defined as:

Definition 5 (k-minority)**.** *A k-minority of acceptors is a set of k-acceptors which is not a k-quorum.*

Figure 7.1 sketches the execution of PRBR during normal operation: A client wants to execute a (read or write) command c for the item identified by key k. It sends a message to any proposer p with content c and k. Depending on k, p will now exchange messages with the corresponding k-acceptors in the system. The algorithm executed here is similar to basic Paxos as described in Section 4.3 and will be described in Section 7.2 in detail. Each k-acceptor acts as an instance of a replicated state machine which only contains the value of key k. As part of the exchange with p, any k-acceptor which has voted for c will apply the command to the value. Once a k-quorum has voted for c it is considered to be chosen as the next command in the consensus sequence. Only then it is possible for p to learn the result of the command. Once p has learned the result, it notifies the client.

As long as p is able to keep track of multiple concurrent proposals, it can establish multiple commands for different keys at the same time without interference. Of course, one or multiple proposers might try to concurrently establish a command for the same key. In this case, the proposals will create a conflict that has to be resolved first before the respective k-acceptors are able to choose a command.

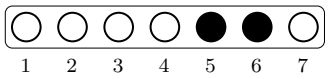
Note that there is no separate learner process involved. Due to the design of PRBR, the roles of learner and proposer cannot be easily detached from each other. Therefore, each proposer in PRBR functions as a learner at the same time.

7.1.2 Consistent Quorums

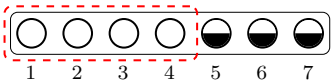
Before the command sequence of a given item can be extended by a new command proposed by a proposer p, it is necessary for p to know whether the currently newest command in the sequence was chosen or not. Let c_{prop} be the command p wants to propose, c_{new} the most recent command at least one k-acceptor has voted for, and c_{prev} the predecessor of c_{new} in the consensus sequence.

If a k-quorum has voted for c_{new}, then it is chosen. In this case, c_{prop} must be proposed as the command following c_{new} in the sequence. Otherwise, the stability requirement of consensus sequences would be violated. If, however, c_{new} was not chosen yet, then p is allowed to propose c_{prop} as the follow-up to c_{prev}. In fact, depending on the messages p receives, it might not even know of the existence of c_{new}.

The following example might help to understand this better: Consider a setting in which an item with key k is replicated seven times. Therefore, the system contains seven k-acceptors. The state of a k-acceptor who has not voted for c_{new} yet, i.e. the old state, is denoted by ○. ● is used for k-acceptors who have already voted for c_{new}. Consider the following configuration of k-acceptors:



Only the k-acceptors numbered 5 and 6 have voted for the new command. A k-quorum has a size of at least four, because a proper majority of k-acceptors is needed. Therefore, c_{new} has not been chosen yet. p can read any given k-quorum as part of the protocol. Naturally, p only knows the state of a k-acceptor if it is part of this quorum. The state of all other k-acceptors is unknown. An unknown state is denoted by ◒. Assume p now reads k-acceptors 1-4:

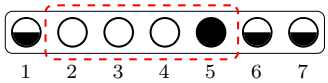


p has received a k-quorum with state ○. This is called a *consistent* k-quorum.

Definition 6 (Consistent k-Quorum)**.** *A consistent k-quorum is a k-quorum Q, for which all k-acceptors $q \in Q$ have the same state.*

p now knows that command c_{prev} was chosen as part of the sequence, but it does not know if there exists a k-acceptor which has voted for a newer command. However, p can be certain that c_{prev} is the newest chosen command because at most three k-acceptors can be in state ●, which is a k-minority. p can therefore safely propose its new command c_{prop} as successor of c_{prev}.

Another possibility for p is it to read k-acceptors 2-5. In this case, p sees the following state:



The state p sees in this case is not consistent. It is possible for both ○ and ● to exist in a k-quorum. Since ● might exist in a k-quorum, p cannot propose

c_{prop} as follow-up for c_{prev} without risking to violate the stability requirement. If ○ exists in a k-quorum, a concurrent proposal from another proposal could read this k-quorum and propose a different command on top of c_{prev}. Therefore, p can also not safely propose its command as successor of c_{new}. p's only option when reading an inconsistent state is it to try help ● reaching a k-quorum. Only then can p safely propose its own command. How this is done is described further in Section 7.2, where the algorithm of PRBR is discussed in detail.

7.1.3 Commands

Commands are structured data send by clients to the consensus system. In the context of PRBR, a command consists of so-called *filters*. Every filter is a function that is applied at a certain point during the execution of the protocol. Furthermore, commands are issued for a specific key.

Definition 7 (Command)**.** *A command is a 3-tuple* $\{k, v_{write}, Filters\}$*, where* k *is the key of the item accessed or modified by this command,* v_{write} *the value to write, and* $Filters$ *an n-tuple of filters.*

In most cases, v_{write} does not simply indicate the new value that will be stored in the acceptors after the command was applied. Commands can operate on only parts of the existing value and use v_{write} as an additional argument. For example, the filters of a command can be used to model an *append* operation. Here, v_{write} would indicate the element that would be appended. The following three types of filter exist in PRBR:

ReadFilter In PRBR's equivalent to Paxos phase 1, acceptors apply a ReadFilter on their currently stored value val. The result v_{read} is used for the remainder of the protocol instead of val. If values consist of multiple elements, for example when using Scalaris' transaction protocol, the ReadFilter can be used to retrieve only the needed parts of the value, e.g. the locks. If the whole value is needed for the execution of a command, then $val = v_{read}$. Then the ReadFilter is called a *no-op*. In use cases where values are large, ReadFilters can significantly reduce the size of the exchanged messages. The application of a ReadFilter does not modify the stored value of the respective acceptor.

ContentCheck Once a proposer has received a consistent k-quorum in the equivalent to the beginning of Paxos phase 2, the ContentCheck is used to verify if this command is a valid successor to the latest established consensus based on v_{read}. As continuation of the transaction protocol example, the ContentCheck can test if the required locks are present to modify the value. If the ContentCheck fails, the protocol can terminate early and the proposer can notify the respective client that its command is not valid. A ContentCheck returns a 2-tuple: A boolean value indicating whether the check was successful or not, and so-called UpdateInformation i that can provide additional information to the WriteFilter.

WriteFilter Once an acceptor has voted for a command, it applies the WriteFilter of the command. Based on val, v_{write} and i, the WriteFilter returns a new

value v_{new}. The acceptor stores v_{new} as the new state of this replica. In addition to v_{new}, the WriteFilter returns a return value v_{ret} that is send to the proposer and which in turn sends it to the respective client.

In contrast to Multi-Paxos, PRBR differentiates between *read* and *write* commands. This distinction allows PRBR to terminate the protocol early when processing read commands.

As implied by its name, a read command will return the value of given key (or part of it) without modifying it. Since no modification takes place, reads do not need v_{write}, a ContentCheck or a WriteFilter. Let $\perp$ denote an empty or non-existing value.

Definition 8 (Read Command)**.** *A read command is a command with* $v_{write} = \perp$ *and* $Filters = \{rf\}$*, where rf is a ReadFilter.*

A write command modifies the value and returns the result of the modification to the client. Therefore, a write command uses all three types of filter.

Definition 9 (Write Command)**.** *A write command is a command with* $Filters = \{rf, cc, wf\}$*, where rf is a ReadFilter, cc a ContentCheck and wf a WriteFilter.*

For conciseness, c_e denotes element e of command c. For example, c_k denotes the key and c_{rf} the ReadFilter of c. By convention, the abbreviations rf, cc and wf are used for ReadFilter, ContentCheck and WriteFilter, respectively.

7.1.4 Rounds

Rounds in PRBR are the equivalent to Paxos' proposal numbers. Similar to Paxos, rounds are required to be unique to distinguish messages regarding different proposals. This is achieved by defining rounds to be 2-tuples, consisting of a round number and a round ID. The round number is a non-negative integer, whereas the composition of the round ID can be chosen arbitrarily depending on the implementation, as long as its uniqueness can be guaranteed. For example, one valid strategy could be to include the proposer's process ID in the round ID, to ensure that no other proposer can generate the same round ID.

Two rounds a and b are equal if both their round number and round ID match. a is only greater than b, if a's round number is larger than b. If their round number matches, but their round ID not, then $a \neq b$ and $a \not> b$.

7.2 The Algorithm

PRBR can be divided into three phases: round_request, read and write phase. The read and write phase are quite similar to the phases of Paxos, whereas the round_request is an optimization of the read phase by letting acceptors chose the round number for a proposer. This section covers all phases in detail and is followed by some example executions of PRBR. Understanding any non-trivial algorithm solely by reading its textual description or pseudocode is difficult. Therefore, some executions examples will be provided in Section 7.3.

Internal State of Acceptors

Every k-acceptor holds four state variables.

k The key of the item this acceptor is responsible for.

val The current value of the item this acceptor is responsible for. If no value exists yet, then $\perp$ will be used to denote a nonexisting value.

$\mathbf{r}_{read}$ The highest round in which a ReadFilter was applied to *val*. As an initial value any round can be chosen that is smaller than any round a proposer would choose. For example, the round number can be set to 0 and an arbitrary round ID part can be used. Will be also referred to as read round.

$\mathbf{r}_{write}$ The highest round in which a WriteFilter was applied to *val*. It has the same initial value than r_{read}. Will be also referred to as write round.

Note that there does not exist an explicit command log. As soon as an acceptor votes for a command, it applies the respective WriteFilter immediately. The command sequence is established implicitly by the sequence of values an acceptor holds during its lifetime.

7.2.1 Phase 1: round_request

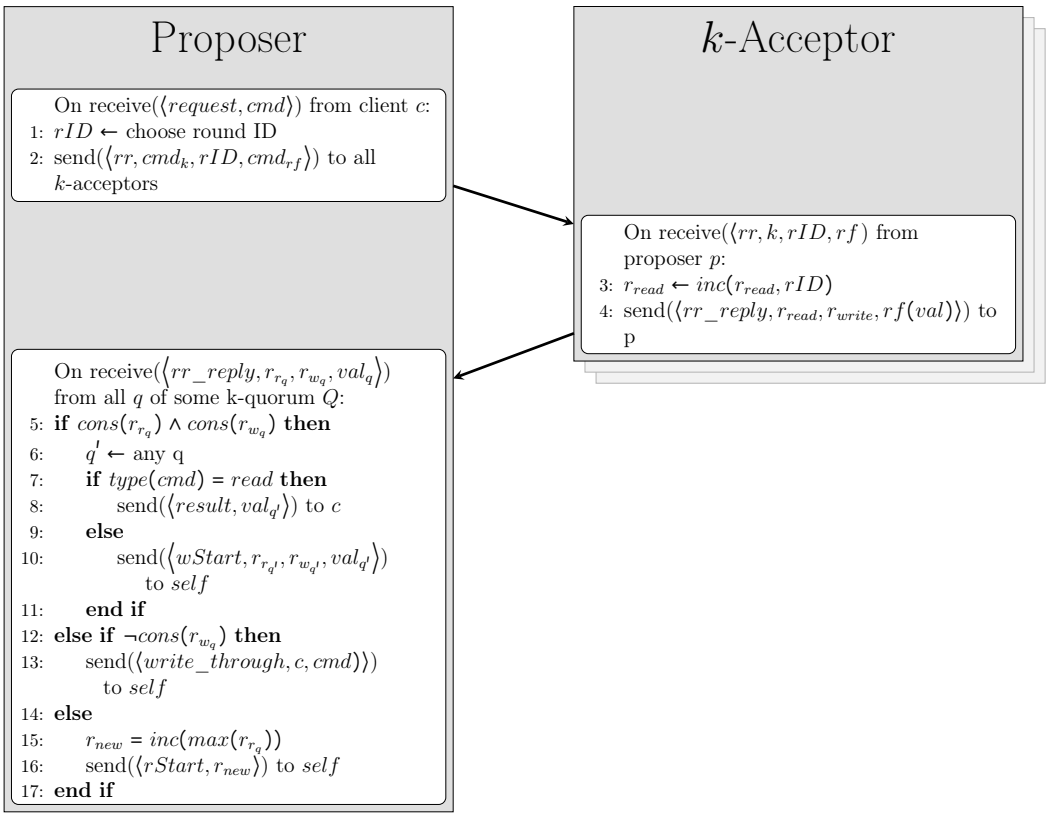
The *round_request* phase is the first phase and entry point of PRBR. It starts as soon as a proposer p receives a *request* message with a command *cmd* from client c. As described earlier, *cmd* consists of k – the key of the item which will be accessed by the command – and a list of filters. It is assumed that the proposer has some means to remember *cmd* and c across its actions and knows when this information can be removed safely. Code Listing 7.1 depicts the pseudocode of the round_request phase and can be used as a reference for the textual description.

In basic Paxos, the first phase begins with p choosing a round number (proposal number in Paxos terminology). However, how does p know what round number is high enough to be successful with its proposal? If p chooses a round number which is too small, then it will only get denies from acceptors and has to try again, thus wasting the time of two message delays. Therefore, PRBR lets the acceptors choose the round number as part of this phase.

The proposer begins by choosing only the round ID part of the round, *rID*. To reiterate, it consists of p's PID and some unique request ID. Afterwards, p sends the round ID along with the key and the ReadFilter of *cmd* to all k-acceptors (line 1-2).

Upon reception of the message, each k-acceptor increments its read round r_{read} and replaces the round ID part with *rID* (line 3). In Paxos terminology, the acceptor has now promised to not vote for a new command with a round smaller than the new r_{read}. If there are any proposals still in progress for the old value of r_{read}, then the acceptor will deny them. The acceptor now applies the received ReadFilter on *val* and sends the result to p along with both of its rounds r_{read} and r_{write} (line 4).

The second goal of this phase is to verify if there is still an unfinished proposal in progress. Once p received a reply from a k-quorum of acceptors, it checks if this k-quorum is consistent. For that, p checks if all received read rounds and all



Code Listing 7.1: Pseudocode of PRBR's round_request phase.

received write rounds are consistent. Note that p does not have to compare the received values for consistency. As will be shown later, two k-acceptors with the same write round will always have the same value (see property P3). p's behavior depends on whether the quorum was consistent and the type of command received by c:

(1) First, consider the case of a consistent k-quorum (line 5-11).

(1.1) If *cmd* is a read command (line 7-8), then p does not have to continue with the next phases of the protocol and can deliver the received value to the client. That way read commands are not explicitly included in the command sequence – the command is not chosen by k-acceptors in the same manner a write command would be. Therefore, the read command will not be appended to the command sequence and thus will not be learned by a proposer. This means the safety requirements established in Section 5.2 are not applicable. However, read commands must still adhere to strong consistency. To be more precise, this means that a read command is never allowed to return an older state than any previous, non-concurrent command.

P 1. *Once a read command has returned some result based on a command sequence seq, all subsequent reads return a result based on a command sequence with seq as prefix.*

Since writes also return results to clients, they must be considered here as well. Once a write has been completed, all subsequent read commands must return the

new state. However, the state modifications done by writes are handled in the write phase of PRBR. Therefore, some knowledge must be presumed here:

The write phase behaves similar to Paxos' second phase. Most importantly, a k-quorum must vote for a write command in the same round before it can be learned. In other words, a write command must be chosen before its result can be delivered to a client. This will be further discussed in Section 7.2.3.

This means that the round_request phase can return the result of read command at this point in the protocol, if the following property is satisfied:

P 2. *Once a write command cmd_w has been chosen, all subsequent reads include cmd_w.*

Proof sketch. It was already established in Section 7.1.2 that no newer chosen command can exist if p reads a consistent k-quorum, because this implies the existence of two k-quorums with an empty intersections. Therefore, p can safely return the result here without violating P2.

P1 holds if no read returns an older state than a previous read command. Because p has seen a consistent k-quorum, all later reads made by any proposer will see at least one k-acceptor in this state (or a newer state) due to the k-quorum property. Therefore, any later read will either (a) see the same (or newer) state as a consistent quorum, or (b) see an inconsistent state. In the case of (a), the read result can be returned without violating P1. In the case of (b), the proposer does not return a result. This will be covered in (2.1). □

(1.2) If *cmd* is a write command (line 9-10), then p can proceed to the write phase of the protocol. It does so by sending a *wStart* message to itself which contains the seen read round, write round and value. Again, due to the consistent k-quorum, p can be certain the seen state can be safely used as the basis for a new command.

(2) Next, consider the case of an inconsistent k-quorum (line 12-17).

(2.1) If the received write rounds are inconsistent (line 12-13), then p is in the same scenario as described in Section 7.1.2. p can not propose its command safely (and can also not return a read without violating P1). Because the original proposer of the command might have crashed, p can also not wait until someone else has finished the started consensus. Therefore, p starts a repair process called *WriteThrough*. Basically, a WriteThrough uses a special command that reads the complete value (and thus effectively reading the complete command sequence) and overrides any existing state with its WriteFilter. Afterwards, p re-initiates its original request. The specifics of WriteThroughs are covered in Section 7.2.4.

(2.2) If the received read rounds are inconsistent (line 14-17), then either (a) a read is in progress or (b) a write is in progress but has not established a full consensus yet. For (b), the proposer has no way of knowing if there already exists a k-acceptor which has voted for the new command. In contrast to (1.2), p cannot simply propose its own command even if it is clear that no newer command can be chosen:

Every k-acceptor which has received a message from p increments its read round. If the read rounds are inconsistent before the increment, then chances are that they are inconsistent after they have processed p's message. Let $r_{highest}$ be the highest read round seen by p. Due to the inconsistencies, it is not certain that

there is a k-quorum with $r_{highest}$ as its read round. Therefore not enough acceptors might have given a valid promise.

To solve this, p starts a read with an explicit round number. The round number p chooses is $r_{highest}$ incremented by one. This way p can be sure that a k-quorum exists that can give a promise using this round (under the assumption that no concurrent command exists). This is done in the *read phase* of PRBR, which is described in the following section.

7.2.2 Phase 2: read

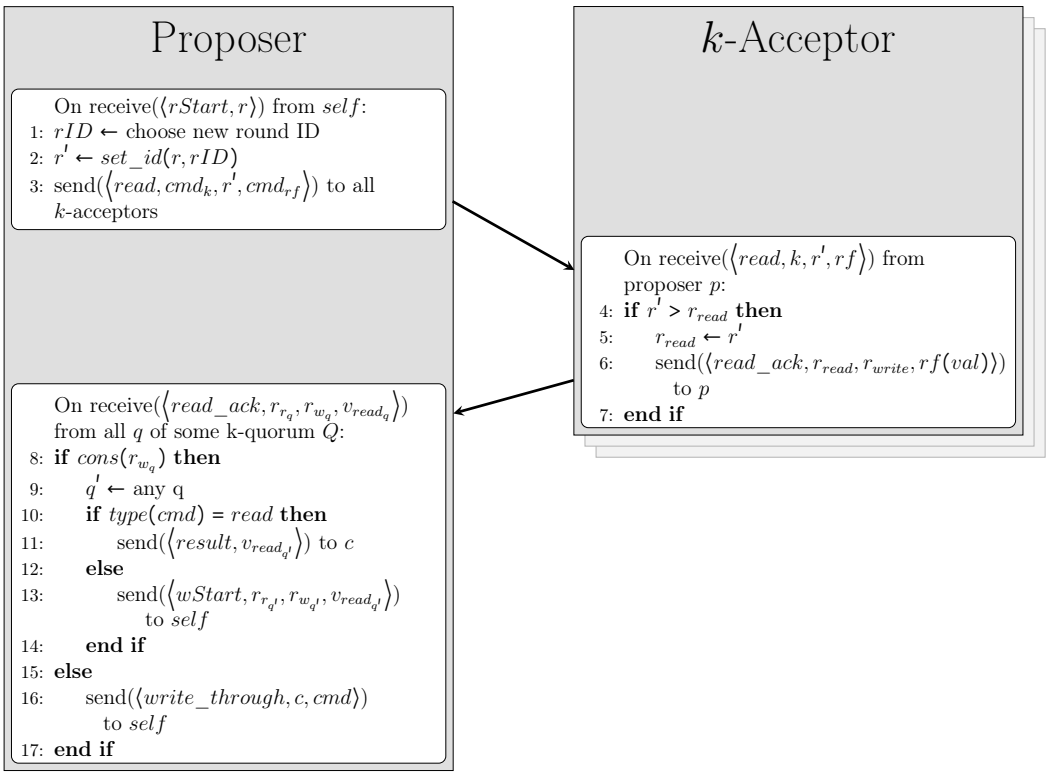
The read phase will only be executed if there are concurrent commands for the same key, or some messages of a previous proposal were lost or delayed. The phase fulfills a similar purpose to the round_request phase, with the exception that p now knows a round which it can choose to likely succeed with its proposal. Thus, the structure of the round_request and read phase are similar, as can be seen by comparing the pseudocode of the round_request phase (Code Listing 7.1) with the pseudocode of the read phase (Code Listing 7.2).

The phase starts with proposer p receiving a *rStart* message from itself. The message includes a round r, which is the highest read round received in the k-quorum of the previous phase incremented by one. It is assumed that p still remembers the original request, thus having access to c and *cmd*. Alternatively, this information can be included in the *rStart* message.

First, p creates a new round r' with the same round number as r but a newly chosen round ID (line 1-2). This is a preventive measure because there might already exist some k-acceptors with r as read round. To provide a simple example, assume that three k-acceptors exist with read round numbers $1, 2, 3$, respectively. This can happen since an arbitrary number of round_requests can be executed concurrently. During the execution of the round_request phase, p reads the first two k-acceptors. Now their read round numbers are $2, 3, 3$. p saw an inconsistent k-quorum and proceeds to the read phase with a round with round number 4 (due to the increment). However, a delayed message from p's round_request can still reach the third k-acceptor. Now, this acceptor also has a read round with round number 4. If p would not change the round ID in the following read phase, then this new request's round would be indistinguishable from an already existing round. To prevent potential problems for certain interleaving of messages when handling concurrent proposals caused by some unknown subtlety of the protocol, p therefore changes the round ID.

p then sends a *read* message to all k-acceptors with the key and ReadFilter of the command submitted by c and the new round r' (line 3). Each k-acceptor receiving the message behaves in a similar manner than in basic Paxos during phase 1. If the acceptor's read round is higher than the received round, then it denies the request. Note that the pseudocode does not depict deny messages for better readability. Deny messages include the acceptor's read round.

If r' is larger than r_{read}, then the k-acceptor sets $r_{read} = r'$, thus preventing any proposals with smaller rounds from being accepted by it in the future. Afterwards, the acceptor applies the received ReadFilter and replies to p with r_{read}, r_{write} and the result of the ReadFilter in a read_ack message (line 4-7).



Code Listing 7.2: Pseudocode of PRBR's read phase.

p waits until it has received a k-quorum of read_acks. Here, p might also realize that its request failed due to receiving too many denies. To be more precise, p knows that its request failed if it has received a k-quorum of denies. In this case, there is only a potential k-minority of acceptors left from which it could receive read_acks. p therefore has to restart the read phase with the highest received round in deny messages.

Once p has received the k-quorum of read_acks, it is presented with similar options than in the previous phase. First, p checks if the seen k-quorum is consistent. For that, it checks if all received write rounds are the same (line 8). It is not necessary for p to compare read rounds since they are equal to the round sent to the acceptors in the beginning of this phase.

If the k-quorum is inconsistent (line 15-17), this means a write is in progress and at least one k-acceptor has voted for this command. As with the previous phase, p has no way of knowing if the new command was already chosen or not. Thus, p has no other option than starting a WriteThrough (see Section 7.2.4).

If the k-quorum is consistent (line 9-14), then p can deliver the result of the ReadFilter to the client c if *cmd* is a read command. Otherwise, *cmd* must be a write and p can proceed to the write phase by sending a *wStart* message to itself. This message includes the same information than in the round_request phase: the seen read round, write round and the read value.

Due to the similar structure of the round_request and read phase, an identical argument for p's decision can be applied here.

7.2.3 Phase 3: write

The write phase starts with p receiving a *wStart* message from itself. In addition to the message's type, it contains three elements r_r, r_w and v_{read}. To summarize the previous phases, p can infer the following information from the content of the message:

r_r The round used by p to receive promises from a k-quorum. A least a k-quorum has $r_{read} \geq r_r$, because an acceptor never decreases its rounds. If there is no concurrent request for the same key, then at least a k-quorum has $r_{read} = r_r$. The remaining k-minority of acceptors can have either a higher or lower read round. As will be explained in this section, the write phase follows the same basic principles as basic Paxos. Therefore, no write will succeed with a round smaller than r_r.

r_w The write round of the latest chosen command. There cannot be a chosen command with a higher write round r'_w, since this would imply that a k-quorum with $r_{write} = r'_w$ exists. Then p would have read an inconsistent k-quorum in an earlier phase because any two k-quorums have a non-empty intersection. However, p proceeds only to the write phase after receiving a consistent k-quorum. Thus, assuming the existence of a higher chosen command leads to a contradiction.

v_{read} The result of $cmd_{rf}(val)$, where val is the state of the item with key k in write round r_w. As already mentioned, all k-acceptors with the same k and r_w always have the value for val (see property 3). Since r_w is the write round of the latest chosen command, val is the most recent state of the item. No client has received a result which is based on a newer state yet.

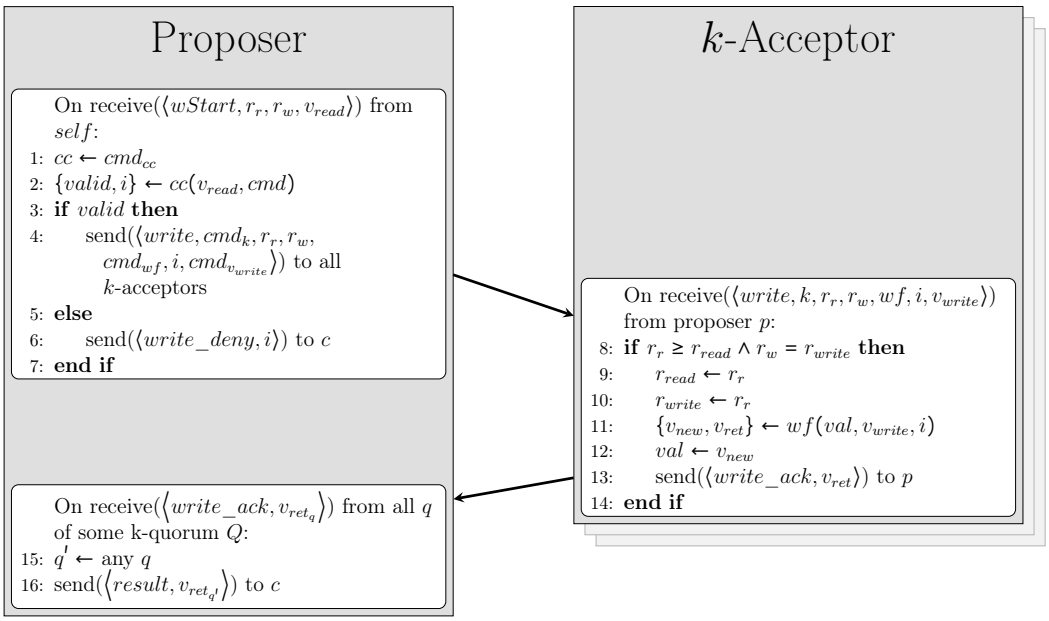
Analogously to the read phase, it is assumed that p still has access to c and *cmd*: The client and its proposed command, respectively. *cmd* must be a write command because all reads are delivered in an earlier phase.

At the beginning of the write phase, which is depicted in Code Listing 7.3, p uses the ContentCheck of *cmd* to verify if this write command is a valid successor to the current state of the consensus sequence (line 1-2). For example, when using Scalaris's transaction protocol, the ReadFilter of *cmd* might have read the write lock of an item. If the correct transaction has acquired the lock, *cmd* is a valid command. Otherwise, the transaction is not allowed to manipulate the value, in which case the command will be denied.

If the command is not valid, p notifies the client with a *write_deny* message, which includes a reason for the deny given by the ContentCheck (line 6). If the command is valid, then p sends a *write* message to all k-acceptors. The message includes the key, WriteFilter, and write value v_{write} of *cmd*, as well as both rounds r_r, and r_w. In addition, the UpdateInformation of the ContentCheck are included as well (line 4).

Every k-acceptor receiving the message checks if it can vote for the received WriteFilter (and by extension *cmd*) in round r_r. For that, two conditions must be fulfilled: **(1)** $r_r \geq r_{read}$ and **(2)** $r_w = r_{write}$ (line 8).

If both conditions are satisfied (line 9-13), the acceptor sets both read round and write round to r_r (the read round is updated because this acceptor might not



Code Listing 7.3: Pseudocode of PRBR's write phase.

have received the messages of the previous phases). Then it applies the received WriteFilter. Recall from Section 7.1.3 that WriteFilter return two values v_{new} and v_{ret}. v_{new} is the new state of this acceptor, whereas v_{ret} will be returned to the proposer. Depending on the WriteFilter, both values might be identical. However, v_{ret} is simply an *ok* or *true* in many cases that indicates that the command was applied successfully. Once val is set to v_{new}, the acceptor has voted for the command. It then sends a *write_ ack* message to the proposer and includes v_{ret}.

Is (1) or (2) not satisfied, then the acceptor does not vote for this command and sends a *write_ deny* message to p.

Condition (1) is analogous to basic Paxos' second phase. It prevents an acceptor from voting for a command with a lower round than it has already given a promise for in previous phases. Condition (2) has no such equivalent because here acceptors vote for a growing sequence of commands in-place (i.e. without allocating new resources for each new command). (2) is necessary to ensure that all acceptors apply all commands without omissions. The following scenario illustrates how omissions of commands might happen:

Assume an item that is replicated three times is used to store a list. An *append* command is provided to clients to append elements to the list. Three append commands are issued sequentially to append element A, B and C, respectively. They will be referred to as cmd_A, cmd_B and cmd_C.

All k-acceptors complete the round_request phase and vote for cmd_A. Since a k-quorum has voted for the command, cmd_A is now the newest chosen command.

Afterwards, a client submits a request with cmd_B. But one of the k-acceptors is unavailable. This can happen due to various reasons: The acceptor might have crashed, links might have lost messages, or the relevant messages might simply be delayed for a long time. However, two k-acceptors are a proper majority and thus a k-quorum. The protocol can proceed normally and both available k-acceptors vote for cmd_B as the next command in the consensus sequence. cmd_B is therefore

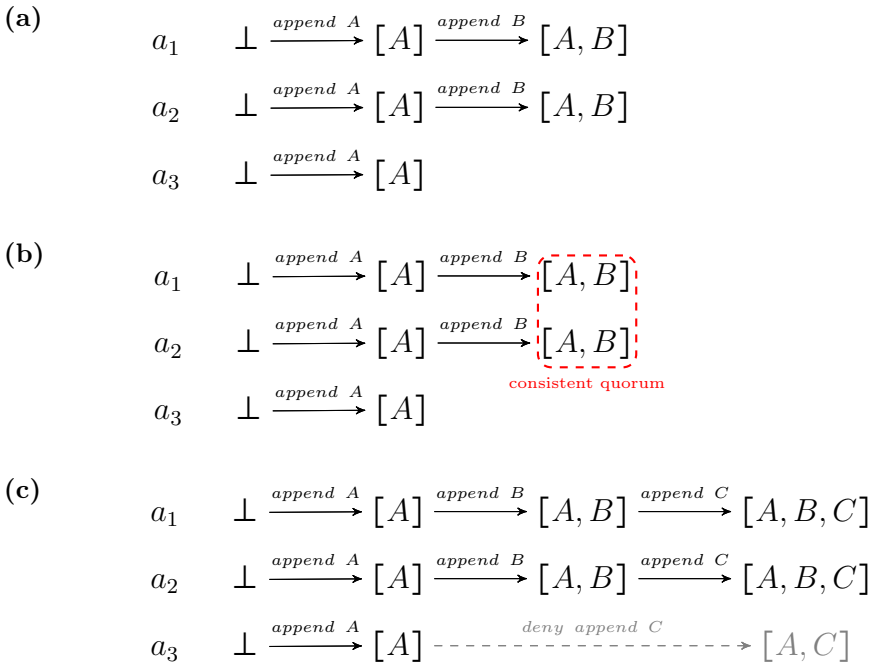


Figure 7.2: State of k-acceptors with sequential append commands.

chosen. This state is depicted in Figure 7.2a.

Now, the third command cmd_C is submitted to a proposer. In every step of the protocol, proposers send their messages to all k-acceptors, but wait only until they receive responses from a k-quorum. Then the proposer acts immediately because it does not know if more correct k-acceptors exist. All messages arriving later are irrelevant. As shown in Figure 7.2b, the proposer receives a consistent quorum if the replies of a_1 and a_2 arrive first. Because of that, the proposer can proceed to the write phase of the protocol. As with the previous command, a_1 and a_2 vote for the cmd_C, thus choosing the command. However, a_3 can also receive the *write* message. If a_3 has not received any additional messages, its read round cannot be higher than the read round of a_1 or a_2. Therefore, condition (1) is satisfied. But condition (2) is not. Round r_w included in the *write* message is that of cmd_B because a_1 and a_2 voted for this command. a_3's write round must be equal to the round cmd_A was chosen in. Since rounds are unique for every request they cannot be the same. Thus, a_3 does not vote for cmd_C.

Without condition (2), a_3 would have voted for cmd_C. Since it skipped the second command, a_3's value would be $[A, C]$, whereas the other replicas value is $[A, B, C]$. Since the last command all k-acceptors voted for is the same, their write round is the same as well. This would be problematic. Any subsequent read would see consistent rounds and therefore might deliver either $[A, C]$, or $[A, B, C]$. This clearly violates the stability requirement. Even if the protocol explicitly compares the values and detects an inconsistent state, the proposers have no means of knowing which value encodes the complete sequence. This leads to the following property the protocol must satisfy:

P 3. *Any two k-acceptors that vote for a command in the same write round* r_{write} *hold the same value val.*

Proof sketch. Property P3 can be shown by inductive argument. As the base case, all k-acceptors are initialized with the same write round and value. Due to condition (2), all k-acceptors that vote for the same $(n+1)^{th}$ command have also voted for the same n^{th} command in the same write round. Since only a single proposer sends *write* message for a specific command and all those message contain the same rounds, all k-acceptors that vote for the same $(n+1)^{th}$ command have the same write round. Furthermore, two different commands cannot be proposed using the same write round since every proposer chooses a unique round ID at the beginning of the protocol.

□

It is important to note at this point, that the update of an acceptors' state variables (line 9,10 and 12) within its action in the write phase must occur atomically. Otherwise, an acceptor that crashes and recovers at the wrong time might only update part of its state. For example, it might update its write round but crashes before its value can be updated, which can cause a violation of property P3.

Once p has received a k-quorum of *write_ ack* replies it has learned the result of *cmd* (line 15-16). In other words, p learned a sequence of write commands with *cmd* as the last element in this sequence. Due to property P3, all k-acceptors that have voted for the command return the same result. Thus, p can forward the result to c. This concludes the protocol.

It is also possible for p to receive a k-quorum of *write_ deny* messages. p knows that its write failed since only a k-minority remains that can vote for the command in this round. Therefore, p has to retry its proposal in a higher round. For that purpose, it is useful to include the reads rounds of the acceptors sending denies. That way, p can use the highest received round to retry starting with the read phase.

Analogously to reads, it must be shown that writes satisfy the safety requirements for consensus sequences.

Nontriviality is easy to see. Acceptors only vote for commands that have been proposed by a proposer and proposers propose only those commands that have been send by a client. Since Byzantine failures are not considered, no process deviates from the protocol and the content of messages are not corrupted. Therefore, a command is only included if it has been proposed before.

Stability requires that a fixed proposer (which doubles as a learner in PRBR) always learns an increasing sequence of commands. Once a command is included in the learned sequence, it must be included whenever the proposer learns the sequence in a higher round. Since a command can be learned only after it was chosen, PRBR must exhibit the following property:

P 4. *Once a write command cmd_w has been chosen, it is included in the command sequence of all write commands chosen in a higher write round.*

Proof sketch. Once a write command cmd_w has been chosen, a k-quorum has voted for it. A k-acceptor never discards its vote for cmd_w[7]. – all subsequent write

[7]It will be shown later that a k-acceptor, in fact, can discard its vote for a command as part of a WriteThrough. However, this is only possible if the command is not chosen.

commands are applied on top of it Therefore cmd_w will be always included in the command sequence of at least a k-quorum. Due to P3, no acceptor can skip a chosen command in its command sequence. Therefore, cmd_w is included in *all* votes of any k-acceptor in a higher write round once it has been chosen. □

In addition to P4, the order of chosen commands must not ever change to fulfill stability.

P 5. *Once write command cmd_w has been chosen as the newest element in command sequence seq, the command sequence of all later chosen write commands contains seq as prefix.*

Proof sketch. Due to P4, once a command is chosen it will be never removed from any command sequence it is included in. The only manipulation of a command sequence performed as part of the protocol is to append command at the end of them. Therefore, the order of already chosen commands remains unchanged. □

Consistency is closely related to stability. For any two proposers p_1 and p_2, one proposer must always learn a command sequence which is a prefix of the others learned command sequence. This is a direct consequence of P5, because a proposer learns the state of the command sequence only if its proposed command was chosen.

7.2.4 WriteThrough

The description of the protocol so far focused on the execution path in which the proposer received a k-quorum of consistent write rounds in either the round_request or read phase. However, if there are concurrent write commands for the same key or one of the acceptors recovered from a previous crash, then it is possible for a proposer to receive an inconsistent state.

This problem can be illustrated by the continuation of the example shown in Figure 7.2 on page 35. To reiterate, a sequence of append commands were issued for an item replicated three times. The third k-acceptor did not receive messages of the second command and thus denied the third command. Figure 7.3a shows the state of the k-acceptors at the end of the example. As with the explanation of PRBR's phases, some fixed key k is assumed for the remainder of this section.

When a new command is issued for the item, then the proposer either receives a consistent k-quorum if the replies from a_1 and a_2 arrive first in the round_request phase, or an inconsistent k-quorum if a_3's message arrives earlier. The consistent case is covered in the write phase of the protocol. However, a_3 will never vote for any new commands because it knows that it is missing a chosen command in its sequence.

Eventually, some proposer p will see an inconsistent k-quorum. Even in scenarios in which the communication links to a_3 are systematically slower than all other links, p will definitely see such a quorum if a_1 or a_2 crashes. If this happens, p starts the WriteThrough process by sending a *write_through* message to itself containing the current command *cmd* and the client c that has issued the command. For reference, see Figure 7.1 line 15 or Figure 7.2 line 16 for a proposer triggering a WriteThrough in the round_request or read phase, respectively.

Basically, a WriteThrough is a special write command that attempts to synchronize the states of all k-acceptors. It is executed like any normal write command with two exceptions: First, a WriteThrough proceeds to the write phase even if the proposer receives inconsistent write rounds in a previous phase. Second, k-acceptors will always vote for a WriteThrough if it is in a higher round than the highest round they have given a promise for (which means that they ignore the condition $r_w = r_{write}$ in Figure 7.3 line 8).

Since a WriteThrough is a write, it has a ReadFilter, ContentCheck and WriteFilter. As will be seen later, the write value v_{write} of the command does not matter and is therefore the empty value $\perp$. The filters perform the following operations:

ReadFilter Acceptors do not store their command sequences explicitly. When they vote for a new command, only their stored value *val* will change. Its current value is the result of the sequence of commands they voted for. The ReadFilter of a WriteThrough reads the complete value stored in the respective acceptor ($v_{read} = val$). Therefore, this can be seen as implicitly returning the full command sequence to the proposer.

ContentCheck A WriteThrough is always a valid follow-up command. Therefore, the ContentCheck will always pass. The UpdateInformation i returned by the ContentCheck is the unmodified value v_{read}.

WriteFilter Recall from Section 7.1.3, that a WriteFilter returns a tuple $\{v_{new}, v_{ret}\}$ based on i, *val* and v_{write}. A WriteFilter which is part of a WriteThrough will always return $\{ok, i\}$. This means the new state of an acceptor is independent of its previous state. After it has voted for the WriteThrough command $val = i$. Here, *ok* denotes some arbitrary value that indicated that the command was applied successfully.

A WriteThrough is executed as follows:

Once p has received the *write_through* message, it starts a round_request with the aforementioned write command and itself as the client. It proceeds through the round_request phase (and potentially the read phase if p saw inconsistent read rounds) normally. However, if p receives inconsistent write rounds and consistent read rounds, then it will use the reply of the k-acceptor with the highest write round to start the write phase.

In the context of the example depicted in Figure 7.3, p has received the replies from k-acceptor a_3 and a_2 first. a_2's write round must be higher than a_3's, because a_3's command sequence is a prefix of a_2's sequence. Therefore, p sends the *wStart* message with a_2's read round, write round and value $[A, B, C]$ (see round_request's pseudocode in Code Listing 7.1 on page 29).

In the beginning of the write phase, p applies the content check, which passes and returns the value of the highest previously received reply as UpdateInformation – in this case $[A, B, C]$. p then sends a *write* message as depicted in Code Listing 7.3 (page 34).

As already mentioned, any k-acceptor receiving the message only checks if the command is proposed in a high enough round. Since this command's round is the highest received round in the previous phase, at least a k-quorum votes for the

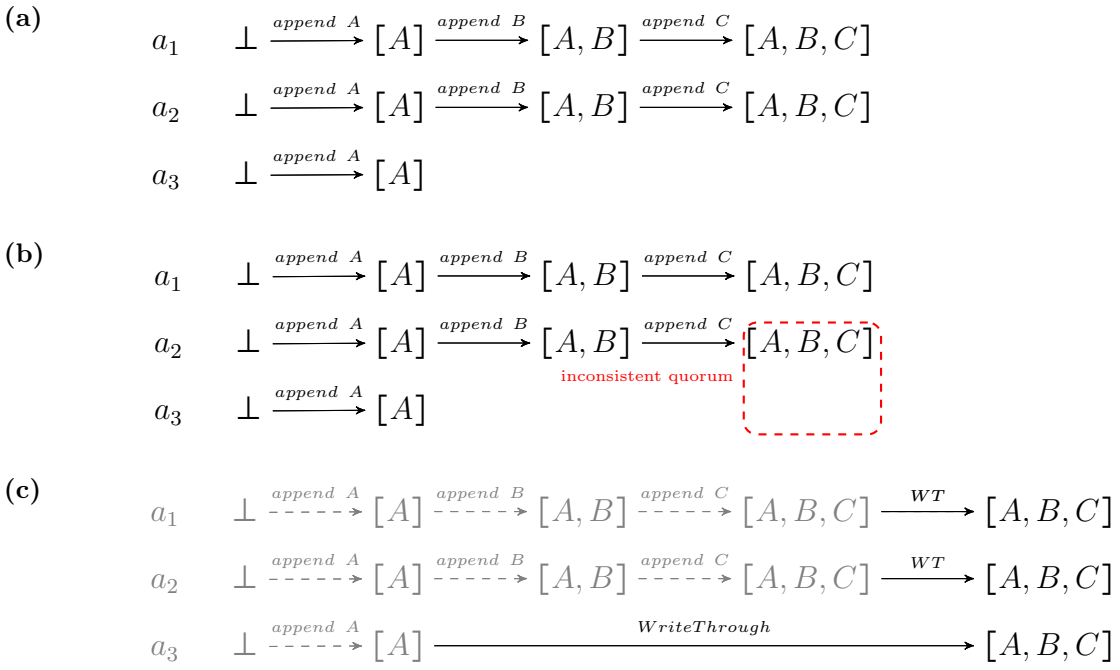


Figure 7.3: Synchronizing k-acceptor state with WriteThrough (WT).

WriteThrough command under the assumption that no concurrent command for k exists. Therefore, the WriteThrough is chosen.

Once p has received a k-quorum of *write_ acks*, it knows that a consistent k-quorum currently exists. The WriteThrough is completed and p can therefore try to establish the original command *cmd* again, as described in the previous sections.

The main idea of WriteThroughs is that the resulting acceptor state is independent of the previous state this acceptor held. This way, the old command sequence of the acceptor is effectively replaced by the command sequence of the acceptor with the highest seen write round.

In addition to the effect seen in Figure 7.3, this can lead to acceptors discarding their vote of their newest command. For example, acceptor a_1 might have already received another command which appends value D. Nevertheless, a_1's value would be still $[A, B, C]$ after the WriteThrough. Since a_1 was not included in the k-quorum, p has no information that a newer command might exist. Due to the WriteThrough a_1 has effectively forgotten its vote for the new command.

However, it is not possible for an already chosen command to be discarded that way. A newest chosen command *cmd* must exist in at least a k-quorum of acceptors. p receives also a k-quorum in the first phase of processing a WriteThrough. The k-quorum property dictates that both quorums have a non-empty intersection. Thus, p receives at least one reply from a k-acceptor that has voted for *cmd*. As a consequence of property P3, a k-acceptor can only vote for a command if it has already voted for all previously chosen commands. Therefore, no chosen command can be discarded due to a WriteThrough, which means that property P4 is not violated. A WriteThrough only copies an existing command sequence to other acceptors. No reordering of commands takes place. Therefore, P5 holds as well.

7.3 Execution Examples

PRBR is considerably more complex than basic Paxos. It might be hard to develop an understanding of the behavior of the protocol with just the textual description. This is especially true without previous experience with other quorum based protocols. Therefore, this section provides some basic examples to highlight some common cases in PRBR. The examples will be illustrated using the following format:

step	*prop.*	*cmd*	a_1	a_2	a_3
1	–	–	$\{0, 0, \perp\}$	$\{0, 0, \perp\}$	$\{0, 0, \perp\}$
2	p_A	cmd_A	$\{1_A, 0, \perp\}$	$\{1_A, 0, \perp\}$	$\{0, 0, \perp\}$

Example 7.1: Example workflow in PRBR.

The figure depicts the state of three k-acceptors a_1, a_2, a_3 for some fixed key k. Each 3-tuple shows the read round (r_{read}), write round (r_{write}) and value (val) of the respective k-acceptor at a specific time. For rounds, 1_A denotes a round with round number 1 and round ID A. The key is not included because a fixed key is assumed.

In every step, each acceptor might receive a message from a proposer (abbreviated as "prop." due to space constraints). The acceptor changes its state accordingly to the protocol and immediately responds to the proposer. Acceptors are highlighted if their response is included in the k-quorum. All surplus messages are ignored by the proposer as per protocol. For the sake of clarity, quorums in the round_request or read phase are colored green (dashed), and quorums in the write phase are colored red (dotted). If an acceptor does not receive any message for any reason (crash, message delay or loss) it is grayed out. The second and third columns are used to distinguish which proposer sent the message to establish which command.

For example, in step 2 of Example 7.1, k-acceptor a_1 and a_2 have received a *round_request* message and their replies are included in the proposers k-quorum, whereas a_3 has not received the message (yet). The message was send by proposer p_A to establish the command cmd_A.

For simplicity, all write commands used in the execution examples are simple append commands that append a single element to a list. This way, the order in which the commands are applied can be easily seen for each acceptor.

Sequential Write and Read

The easiest example is that of sequentially processed commands. Example 7.2 shows the state changes of the k-acceptors in this case.

The example begins with all k-acceptors in their initial state. Some proposer p_A receives a write request to append the value A. p_A starts the round_request phase. In step 2, k-acceptor a_1 and a_2 have received p_A's message, thus incrementing their read rounds and setting the round ID's according to the ID provided by p_A. a_3 is temporarily unavailable and does not receive the message. At the end of step 2, p_A receives the messages from a_1 and a_2 and thus sees a consistent k-quorum.

step	*prop.*	*cmd*	a_1	a_2	a_3
1	–	–	$\{0, 0, \perp\}$	$\{0, 0, \perp\}$	$\{0, 0, \perp\}$
2	p_A	cmd_A	$\{1_A, 0, \perp\}$	$\{1_A, 0, \perp\}$	$\{0, 0, \perp\}$
3	p_A	cmd_A	$\{1_A, 1_A, [A]\}$	$\{1_A, 1_A, [A]\}$	$\{1_A, 1_A, [A]\}$
4	p_B	cmd_B	$\{2_B, 1_A, [A]\}$	$\{2_B, 1_A, [A]\}$	$\{2_B, 1_A, [A]\}$

Example 7.2: k-acceptor states with sequential write and read.

p_A can now proceed to the write phase with $r_r = 1_A$, $r_w = 0$ and $val = \perp$[8]. The ContentCheck passes and p_A sends a *write* message to the k-acceptors in step 3. a_3 is available again and receives the message along with a_1 and a_2. Since both write conditions ($r_r \geq r_{read}$ and $r_w = r_{write}$) are fulfilled for all acceptors, they all vote for the command. The replies of a_2 and a_3 arrive first at p_A. The received state is consistent and p_A can notify c that its write was successful by sending the return value of the WriteFilter to c. The reply from a_1 arrives last and is discarded by p because it already executed its action.

In step 4, a second request was submitted for this key. This time it is a read command. The proposer which has received the request sends the corresponding *round_request* message and receives a consistent k-quorum from a_1 and a_3. The submitted ReadFilter might have returned the whole list, the first element of the list, or just its length. In any case, the proposer can send the read value to the client.

Inconsistent Read Rounds

If two or more commands that modify the same key are submitted to PRBR concurrently, then it is likely that at least one proposer will see inconsistent read or write rounds. In the example shown in Example 7.3, two concurrent write commands cmd_A and cmd_B are submitted which try to append the value A or B to list, respectively. Command cmd_A is handled by proposer p_A and command cmd_B by p_B.

All k-acceptors begin in their initial state. In step 2, p_A sends its *round_request* message to a_1 and a_2. However, the message to a_3 is delayed or lost. Before p_A starts its write phase, p_B's *round_request* messages arrive at a_2 and a_3. p_B therefore receives replies with read rounds 2_B and 1_B, which is an inconsistent k-quorum. Therefore, p_B will continue to PRBR's read phase (step 3) with round number 3.

Afterwards, p_A's write message arrives at a_1. Because this acceptor has not yet received any message from p_B, it votes for cmd_A (step 4). Shortly later, a_3 receives the p_B's delayed *round_request* message. But p_B already received a k-quorum. Therefore, a_1's reply is ignored (step 5).

In step 6, p_B's *read* message arrives at all three k-acceptors. The replies from a_2 and a_3 arrive first. Thus p_B sees a consistent k-quorum and proceeds to its

[8] In these examples, an append command is always a valid follow-up. Therefore, the ReadFilter can always return $\perp$ to minimize the size of the reply messages.

step	*prop.*	*cmd*	a_1	a_2	a_3
1	–	–	$\{0, 0, \perp\}$	$\{0, 0, \perp\}$	$\{0, 0, \perp\}$
2	p_A	cmd_A	$\{1_A, 0, \perp\}$	$\{1_A, 0, \perp\}$	$\{0, 0, \perp\}$
3	p_B	cmd_B	$\{1_A, 0, \perp\}$	$\{2_B, 0, \perp\}$	$\{1_B, 0, \perp\}$
4	p_A	cmd_A	$\{1_A, 1_A, [A]\}$	$\{2_B, 0, \perp\}$	$\{1_B, 0, \perp\}$
5	p_B	cmd_B	$\{2_B, 1_A, [A]\}$	$\{2_B, 0, \perp\}$	$\{1_B, 0, \perp\}$
6	p_B	cmd_B	$\{3_C, 1_A, [A]\}$	$\{3_C, 0, \perp\}$	$\{3_C, 0, \perp\}$
7	p_B	cmd_B	$\{3_C, 1_A, [A]\}$	$\{3_C, 3_C, [B]\}$	$\{3_C, 3_C, [B]\}$
8	p_A	cmd_A	$\{3_C, 1_A, [A]\}$	$\{3_C, 3_C, [B]\}$	$\{3_C, 3_C, [B]\}$

Example 7.3: k-acceptor states with inconsistent read rounds.

write phase. Because a_1's reply arrives too late, p_B has no knowledge that there is an k-acceptor which has voted for command cmd_A.

In step 7, p_B's write messages arrive at all three k-acceptors. a_2 and a_3 vote for cmd_B and reply to p_B accordingly. a_1 denies the write because round r_w in p_B's *write* message is 0 and this does not match with a_1's r_{write} which is 1_A. However, the deny from a_1 does not matter because two k-acceptors already voted for cmd_B. p_B can therefore notify its client that the write was completed.

In step 8, p_A's delayed *write* messages arrive at a_3 and a_2. But due to p_B's intermediate messages, their read and write rounds are too high to vote for the command. Thus, they deny the write and do not change their state. p_A knows that it cannot succeed with cmd_A in this round. Therefore, it retries by starting a new read phase with round number 4. This is continued in the next section.

This example illustrates that the number of possible message interleavings is high, even for a low number of k-acceptors and concurrent requests. In addition, it shows that the conflict created by concurrent commands is costly to resolve. p_A will need at least four round trip times in total to establish cmd_A. The establishment of cmd_B also required an additional round trip.

Inconsistent Write Rounds

This section is the continuation of the previous example. It illustrates how a WriteThrough resolves inconsistent write rounds. To prevent drawing out the example more than necessary, it is assumed that all of p_A's messages arrive at all k-acceptors in the order they were sent and that no concurrent requests exist.

Step 8 depicts the final state of Example 7.3. To reiterate, p_A has received too many denies and is now starting the read phase with round number 4. In the following step, the all k-acceptors receive the *read* message and update their read rounds accordingly. a_1 and a_2 replied first, therefore p_A receives inconsistent write rounds. As per protocol, it must start a WriteThrough.

In general, a WriteThrough behaves like any other write command. This means p_A has initiate another round_request phase. But this time, the ReadFilter of the command returns the full value of the k-acceptors, i.e. the complete list. At the end of step 10, p_A receives read round 5_E consistently but receives inconsistent

step	*prop.*	*cmd*	a_1	a_2	a_3
8	–	–	$\{3_C, 1_A, [A]\}$	$\{3_C, 3_C, [B]\}$	$\{3_C, 3_C, [B]\}$
9	p_A	cmd_A	$\{4_D, 1_A, [A]\}$	$\{4_D, 3_C, [B]\}$	$\{4_D, 3_C, [B]\}$
10	p_A	cmd_{WT}	$\{5_E, 1_A, [A]\}$	$\{5_E, 3_C, [B]\}$	$\{5_E, 3_C, [B]\}$
11	p_A	cmd_{WT}	$\{5_E, 5_E, [B]\}$	$\{5_E, 5_E, [B]\}$	$\{5_E, 5_E, [B]\}$
12	p_A	cmd_A	$\{6_F, 5_E, [B]\}$	$\{6_F, 5_E, [B]\}$	$\{6_F, 5_E, [B]\}$
13	p_A	cmd_A	$\{6_F, 6_F, [B, A]\}$	$\{6_F, 6_F, [B, A]\}$	$\{6_F, 6_F, [B, A]\}$

Example 7.4: k-acceptor states with inconsistent write rounds.

write rounds, thus inconsistent states. As described in Section 7.2.4, p_A uses the reply with the higher write round, in this case 3_C with value $[B]$.

In step 11, p_A sends its *write* message (with $r_r = 5_E$, $r_w = 3_C$ and $i = [B]$). All k-acceptors receive the message and vote for command. The WriteFilter of the WriteThrough overrides the previous values of each k-acceptor. a_1 discards its previous vote for cmd_A, but this causes no problems because cmd_A was never chosen. The WriteThrough is complete. p_A can try again to establish cmd_A. This time p_A is successful and can finally notify the client of the write command's result (step 12 and 13).

7.4 Fast Writes

PRBR's normal execution path requires the completion of at least two phases until a write command can be chosen. An optimization called *fast writes* exists, that allows a proposer to chain multiple sequential writes on the same item. Similar to Multi-Paxos, the first write in such a chain requires at least two round trips, whereas all subsequent writes can skip round negotiation and proceed directly to the write phase. This can be achieved by incrementing the read round of acceptors when they votes for a command using a fast write. By doing so, acceptors behave like they give a promise in a higher round immediately after voting for a write command. An example of fast writes can be seen in Example 7.5.

step	*prop.*	*cmd*	a_1	a_2	a_3
1	–	–	$\{0, 0, \perp\}$	$\{0, 0, \perp\}$	$\{0, 0, \perp\}$
2	p_A	cmd_A	$\{1_A, 0, \perp\}$	$\{1_A, 0, \perp\}$	$\{1_A, 0, \perp\}$
3	p_A	cmd_A	$\{2_A, 1_A, [A]\}$	$\{2_A, 1_A, [A]\}$	$\{2_A, 1_A, [A]\}$
4	p_A	cmd_B	$\{3_A, 2_A, [A, B]\}$	$\{3_A, 2_A, [A, B]\}$	$\{3_A, 2_A, [A, B]\}$

Example 7.5: A proposer chaining write commands using fast writes.

The first write command submitted by p_A is processed normally with the exception that all k-acceptors that have voted for cmd_A increment their read rounds immediately. This allows p_A to skip the round_request phase can directly proposer command cmd_B with round number 2 after cmd_A was chosen.

Fast writes make it possible to write in two message delays. However, they cannot be used in all situations due to the absence of a distinguished leader. In the example shown above, p_A's second write would have been denied if an intermediate request submitted by a different proposer caused an increase of the round numbers. Therefore, this method is only viable if a proposer submits multiple requests in a short period of time or is sufficiently certain that no other request was submitted by a different proposer.

Fast writes slightly change the behavior of acceptors. Mixing normal writes and fast writes in the same examples might cause unnecessary confusion. Thus, fast writes will be used in later sections of this thesis. However, it was important to list this optimization here to give a complete overview of PRBR's capabilities.

7.5 Double Application of Write Commands

PRBR is still work in progress. In its current state, there exist some interleavings in which PRBR's behavior is not ideal and has potential for improvement. Most notably, the way WriteThroughs are handled right now can cause write commands to be applied twice to a value. An example for this is depicted in Example 7.6.

step	*prop.*	*cmd*	a_1	a_2	a_3
1	–	–	$\{0, 0, \bot\}$	$\{0, 0, \bot\}$	$\{0, 0, \bot\}$
2	p_A	cmd_A	$\{1_A, 0, \bot\}$	$\{1_A, 0, \bot\}$	$\{1_A, 0, \bot\}$
3	p_A	cmd_A	$\{1_A, 1_A, [A]\}$	$\{1_A, 0, \bot\}$	$\{1_A, 0, \bot\}$
4	p_B	cmd_B	$\{2_B, 1_A, [A]\}$	$\{2_B, 0, \bot\}$	$\{2_B, 0, \bot\}$
5	p_B	cmd_{WT}	$\{3_C, 1_A, [A]\}$	$\{3_C, 0, \bot\}$	$\{3_C, 0, \bot\}$
6	p_B	cmd_{WT}	$\{3_C, 3_C, [A]\}$	$\{3_C, 3_C, [A]\}$	$\{3_C, 3_C, [A]\}$
7	p_A	cmd_A	$\{4_D, 3_C, [A]\}$	$\{4_D, 3_C, [A]\}$	$\{4_D, 3_C, [A]\}$
8	p_A	cmd_A	$\{4_D, 4_D, [A, A]\}$	$\{4_D, 4_D, [A, A]\}$	$\{4_D, 4_D, [A, A]\}$

Example 7.6: Command cmd_A is applied twice due to WriteThrough.

Proposer p_A proposes cmd_A that appends value A to a list. However, only k-acceptor a_1 votes for cmd_A before proposer p_B receives a different command for the same item and causes all k-acceptors to increment their read rounds (step 3 and 4). This means that p_A will eventually receive two deny messages indicating that its write failed. In the mean time, p_B receives inconsistent write rounds in step 4, thus starting the WriteThrough process. By chance, the WriteThrough includes a_1 in its quorum (step 5) and successfully synchronizes all k-acceptors with this value (step 6). Finally, p_A has received the denies. However, it has no information that a WriteThrough happened and thus proposes cmd_A again (step 7 and 8), which means that cmd_A is included two times in the command sequence.

From p_A's perspective, it is generally not possible to find out if its unsuccessful write was already included in a WriteThrough by examining the received replies in the round_request phase, because it is theoretically possible for an arbitrary number of writes to succeed in the mean time. Therefore, the WriteThrough

mechanism must be modified to reliably notify the original proposers of the commands. Efforts to solve this problem are currently underway.

7.6 Implementation Considerations

Using PRBR in a setting in which n items are replicated r times requires $n * r$ acceptor processes. Typically, n can be quite large, which makes it not efficient to allocate a proper process for each logical acceptor process. However, the protocol is designed in a way that makes it easy to use one proper process to serve as an arbitrary number of acceptors.

Since the logical processes used in this thesis are simply a form of deterministic state machine, it is possible to define the current state of an acceptor and its future behavior for any given input by knowing its state variables. Thus, each proper acceptor process can manage multiple logical acceptors by persisting their states locally, for example in a hash table. Once a proper process receives a message for some key k, it can simply retrieve the state of the respective k-acceptor from the hash table, perform the computations as required by the protocol, and persist the new state of the k-acceptor afterwards.

Naturally, one must be careful to prevent a proper process from handling multiple k-acceptors of the same item. Otherwise, a single process failure would affect multiple replicas.

7.7 Protocol Complexity

As long as all commands for a given key are issued in sequence and a k-quorum of acceptors is available, then proposers will always receive consistent k-quorums in the round_request phase of PRBR. For many systems, it is not unreasonable to assume that this assumption holds for the vast *majority* of requests. As described in Section 5.5, real world systems often only access different keys concurrently. In PRBR, such commands are managed by separate command sequences and therefore do not interfere with each other.

This means that PRBR can handle the majority of submitted read commands in two message delays, whereas four message delays are needed until the result of a normal write command can be delivered. As described in Section 7.4, writes can also be processed in two message delays in certain situations by using fast writes.

Is every item replicated r times, then $2r$ or $4r$ messages will be sent as part of the protocol to handle read or write request, respectively. Note that only messages send between acceptors and proposers are included in this count. In the description of PRBR, proposers sent messages to themselves when transitioning into a new phase. However, this choice was made mainly for an easier separation of the phases and more compact actions. Instead of sending a message to itself, the two actions sending and receiving the local message can be simply merged.

Conflicts can occur every time two or more proposers access the same item. Analogously to basic Paxos, this can potentially lead to dueling proposers, which means that, as a theoretical worst-case scenario, proposers never recover. However, this state is unstable and the probability of long sequences of conflicts asymptotically approaches zero, as already discussed in Section 4.5.

Recovering from inconsistent read rounds requires at least two additional message delays by entering PRBR's read phase. Inconsistent write rounds require a WriteThrough to resolve. Since these behave in general like any other write, at least four additional message delays are needed. This makes conflict resolution expensive. The following chapter, which constitutes the main contribution of this thesis, explores approaches to reduce conflict potential for concurrent access on the same item by making use of the commutative properties of the submitted commands.

8 Weakening Consensus Sequences

PRBR establishes a separate consensus sequence for every key in the system. This makes it possible to handle concurrent requests for different keys without interference, unless some higher-level abstraction, for example a transaction protocol, introduces dependencies between them. In this case, sequential processing of the requests can be enforced via PRBR's ContentCheck mechanism. In comparison to the more conventional approach of managing a single consensus sequence for all keys, as described in Section 5.4, this reduces overhead caused by concurrent requests in settings in which inter-key dependencies are sparse.

PRBR in its current state establishes a total order of commands in regard to a single item. All commands are executed on all replicas in the same order. This means that any two commands for the same key that are submitted concurrently can potentially block each other from succeeding. Such conflicts can lead to dueling proposers or require a WriteThrough to synchronize the state of replicas again. In both cases, considerable overhead in terms of the number of needed round trips occurs.

Often, it is not necessary to strictly order a pair of commands. The most obvious examples are read commands. If there are two concurrent read commands cmd_1 and cmd_2, then the order in which they are executed on a single replica does not matter. Some replicas can execute cmd_1 first, whereas others can execute cmd_2 before cmd_1. Since both commands are reads, they do not modify the state of the respective replica. Thus, they can be executed in either order without changing the result that is returned to the clients. Furthermore, some operations that can be performed on items are inherently commutative. For example, an item might be used to store a set of elements and a command is defined to add elements to set. Naturally, the order in which elements are added to the set does not matter. In such case, replicas can execute the commands in either order with the same end result.

This chapter focuses on weakening the total order in which commands are executed on a single consensus sequence by making use of the commutative properties of commands. Three scenarios will be considered: two concurrent read commands, a read with a concurrent write command, and two concurrent writes.

8.1 Interfering and Commutative Commands

When arguing about the execution order of a set of commands it is useful to introduce the notion of *interfering* commands in addition to commutative

commands.

Let *Cmds* be the set of all possible commands. *Reads* and *Writes* denote the set of all read and write commands, respectively, with $Reads \cup Writes = Cmds$.

Definition 10 (Interfering Commands)**.** *A command cmd_1 interferes ($\asymp$) with cmd_2, if the result returned by cmd_2 depends on the execution of cmd_1 or if the value of any item depends on the execution order of cmd_1 and cmd_2.*

For a more compact notation, the binary relation $\asymp$ will be used. $cmd_1 \asymp cmd_2$ means that cmd_1 interferes with cmd_2. The symbol $\not\asymp$ is used for the negation of interference. $cmd_1 \not\asymp cmd_2$ denotes that cmd_1 does not interfere with cmd_2. Note that $\asymp$ is not symmetric. For example, if $cmd_2 \in Reads$ and $cmd_1 \in Writes$, then $cmd_2 \not\asymp cmd_1$ because no read command modifies the value of an item. However, $cmd_1 \asymp cmd_2$ might hold depending on the choice of cmd_1 and cmd_2.

WriteThroughs are a special kind of write command because they try to synchronize the item's state across replicas and are treated slightly different by PRBR. They act as a synchronization point in the command sequence. Thus, WriteThroughs interfere with *all* other commands.

Two commands *commute* if they can be executed in either order without changing the result of either command. In terms of interference, this can be expressed as:

Definition 11 (Commutative Commands)**.** *A command cmd_1 commutes ($\|$) with cmd_2, if $cmd_1 \not\asymp cmd_2$ and $cmd_2 \not\asymp cmd_1$.*

Two commutative commands cmd_1 and cmd_2 are denoted as $cmd_1 \parallel cmd_2$. $\parallel$ is a symmetric binary relation. Analogously to interference, $\nparallel$ is the negation of $\parallel$. WriteThroughs do not commute with any command because they interfere with all other commands.

8.2 Commutative Reads

During the execution of a read command, the respective item is accessed via the submitted ReadFilter in PRBR's round_request or read phase. The item's value is not modified during the execution of the command.

However, the result of any command depends solely on the current value of the item, since the only operation performed in reads or writes is the application of various filters on top of the value. Therefore, reads do not interfere with any other command.

$$cmd_1 \not\asymp cmd_2 \qquad cmd_1 \in Reads, cmd_2 \in Cmds \tag{8.1}$$

By application of Definition 11, it follows that any pair of read commands always commute with each other.

$$cmd_1 \parallel cmd_2 \qquad cmd_1, cmd_2 \in Reads \tag{8.2}$$

This is a rather unsurprising result. However, it is convenient because it means that the specific read commands that conflict with each other do not matter. This simplifies the modification of the protocol to prevent such conflicts.

8.2.1 Identifying Avoidable Conflicts

Before the protocol can be modified to accommodate commuting read commands, the problems in PRBR's current state must be identified. Example 8.1 shows an example of two conflicting read commands in PRBR's unmodified state.

step	*prop.*	*cmd*	a_1	a_2	a_3
1	–	–	$\{1_A, 1_A, V\}$	$\{1_A, 1_A, V\}$	$\{1_A, 1_A, V\}$
2	p_A	cmd_A	$\{2_B, 1_A, V\}$	$\{1_A, 1_A, V\}$	$\{1_A, 1_A, V\}$
3	p_B	cmd_B	$\{3_C, 1_A, V\}$	$\{2_C, 1_A, V\}$	$\{2_C, 1_A, V\}$
4	p_A	cmd_A	$\{3_C, 1_A, V\}$	$\{3_B, 1_A, V\}$	$\{3_B, 1_A, V\}$

Example 8.1: Concurrent, conflicting read commands.

Due to the message interleaving, the proposers of commands cmd_A and cmd_B both see inconsistent read rounds. Both commands proceed to the read phase with round number 4. However, only one command can succeed in this round. The other command must retry again within a higher round.

In this example, both proposers need at least four message delays before a result can be returned, potentially even more if they block each other in the protocols read phase using ever increasing rounds.

Ideally, they both receive a consistent quorum in the round_request phase and return the result after only two message delays. The easiest way to achieve this is by simply not incrementing the read round of the k-acceptors in the round_request phase when processing read commands.

Change 1. *Acceptors receiving a read command in the round_request phase do not increment their read round.*

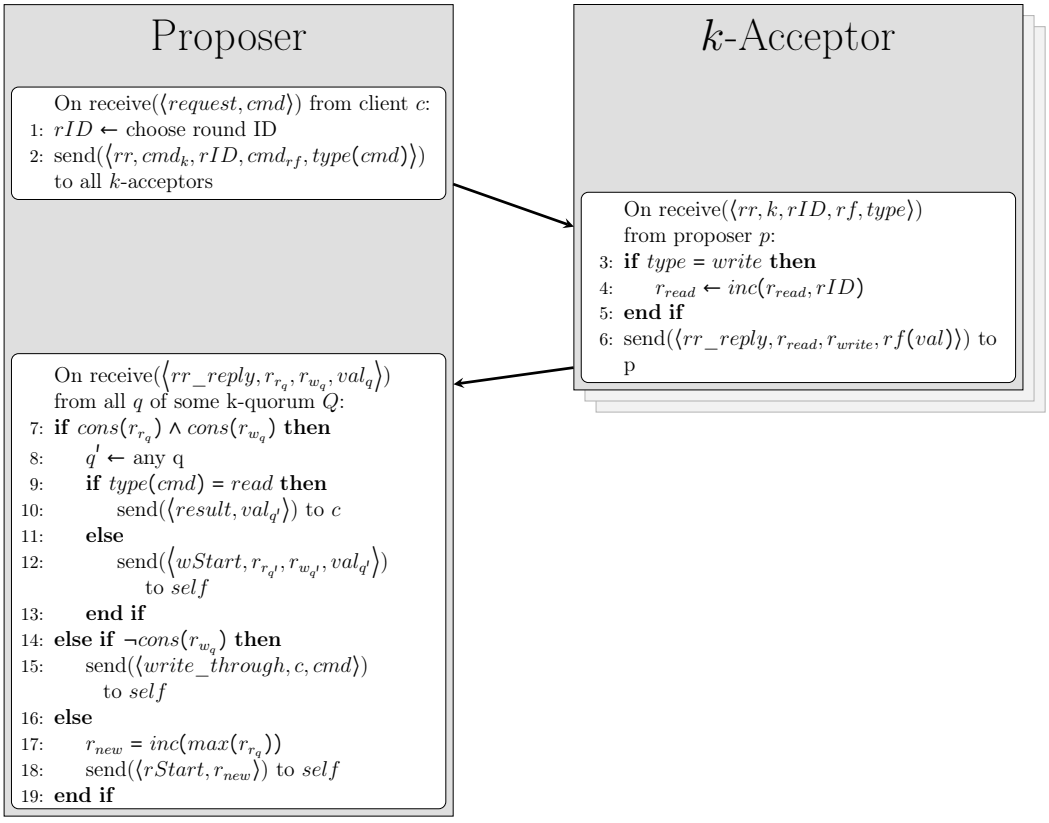
By doing so, an arbitrary number of read commands can access the same item without conflict since the round numbers do not change at all. Proposers always receive consistent quorums. This does not violate property P1 (page 29) since all reads are based on the same command sequence.

The behavior of reads with concurrent writes is unchanged by the modification. Once the command sequence is extended by a new write command (i.e. the write command was chosen), any read will either see inconsistent write rounds and proceed to the WriteThrough phase, or see a consistent quorum and return a result based on the new command sequence. Thus, no read submitted after the write command was chosen returns the old state. Furthermore, no read will return the new state before the write command was chosen because only a k-minority with the new state exists.

Therefore, PRBR still satisfies P1-2 with Change 1 and thus satisfies strong consistency.

8.2.2 Modifying PRBR

As it turns out, the modification required to prevent conflicting read commands is simple. Only the round_request phase must be modified. At the start of



Code Listing 8.1: Pseudocode of PRBR's modified round_request phase.

the phase, the proposer simply includes the type of the received commands in the message before sending it to the k-acceptors. Each k-acceptor receiving the message only increments its read round if the command is a write command. Lines 3-6 in Code Listing 8.1 depicts this change in PRBR's pseudocode.

As long as the command sequence of all k-acceptors is equal and no new write command is started, all proposers executing read commands will receive consistent k-quorums because the k-acceptors' read rounds are not modified. The example at the beginning of this section is now executed as depicted in Example 8.2.

step	*prop.*	*cmd*	a_1	a_2	a_3
1	–	–	$\{1_A, 1_A, V\}$	$\{1_A, 1_A, V\}$	$\{1_A, 1_A, V\}$
2	p_A	cmd_A	$\{1_A, 1_A, V\}$	$\{1_A, 1_A, V\}$	$\{1_A, 1_A, V\}$
3	p_B	cmd_B	$\{1_A, 1_A, V\}$	$\{1_A, 1_A, V\}$	$\{1_A, 1_A, V\}$
4	p_A	cmd_A	$\{1_A, 1_A, V\}$	$\{1_A, 1_A, V\}$	$\{1_A, 1_A, V\}$

Example 8.2: After PRBR's modification, both p_A and p_B receive consistent rounds and can deliver the read.

8.2.3 Impact

The changes made to prevent unnecessary conflicts are small. The first message sent by every proposer now includes the command's type as an additional element. There are only two possible types of commands: reads and writes. This means it is possible to encode this information in one bit (without accounting for the overhead needed to add an additional element to a message).

When processing non-concurrent commands, the number of messages sent and message delays needed by the protocol remains unchanged. Therefore, it is expected that the modification has no measurable impact in this case.

For concurrent read commands, the number of message delays is reduced. All read commands can be processed in exactly two message delays when there is no concurrent write command. Therefore, the throughput is expected to increase.

No additional memory is needed because neither proposers nor acceptors store additional data.

8.3 Reads Commuting with Writes

The second scenario considers a read command cmd_R that submitted concurrently to a write command cmd_W

As already established, $cmd_R \not\leftarrow cmd_W$ for any given pair of commands, because reads do not modify the state of the item. However, the converse cannot be assumed in general. $cmd_W \not\leftarrow cmd_R$ holds only if the write command modifies a different part of the value than returned by the ReadFilter of cmd_R.

For example, assume that both commands are submitted as part of a transaction protocol. Then, cmd_W might set read or write locks and cmd_R's ReadFilter might only read the actual value stored at this key. In this case, the modification made by cmd_W has no impact on the data returned by cmd_R. Thus, $cmd_W \not\leftarrow cmd_R$, which leads to $cmd_W \parallel cmd_R$.

A mechanism to identify whether a pair of concurrent commands commutes or not will be assumed as part of a higher abstraction layer which uses PRBR internally. Doing so at runtime as part of PRBR is not realistic because this would require analyzing the instructions of the filters included as part of each command. In contrast to that, a higher layer typically provides only a finite set of API function for clients to use. Therefore, the set of possible commands is known beforehand which allows it to define any commutative properties between such commands.

8.3.1 Identifying Avoidable Conflicts

As with conflicting reads, an example helps to identify were conflicts between read and write commands can occur and which conflicts can be avoided. Example 8.3 depicts an unfinished write command with three concurrent read commands.

The changes made to PRBR in the previous section are already applied here. Note that the write command cmd_W is no longer blocked by the partially finished read cmd_{R1}. As long as reads are successful in the round_request phase, no acceptor's round numbers are modified. Therefore, such commands are invisible to all other commands. This matches with the fact that read commands do not interfere with any other commands.

step	*prop.*	*cmd*	a_1	a_2	a_3
1	–	–	$\{1_A, 1_A, V\}$	$\{1_A, 1_A, V\}$	$\{1_A, 1_A, V\}$
2	p_{R1}	cmd_{R1}	$\{1_A, 1_A, V\}$	$\{1_A, 1_A, V\}$	$\{1_A, 1_A, V\}$
3	p_W	cmd_W	$\{2_B, 1_A, V\}$	$\{2_B, 1_A, V\}$	$\{1_A, 1_A, V\}$
4	p_W	cmd_W	$\{2_B, 2_B, V'\}$	$\{2_B, 1_A, V\}$	$\{1_A, 1_A, V\}$
5	p_{R1}	cmd_{R1}	$\{2_B, 2_B, V'\}$	$\{2_B, 1_A, V\}$	$\{1_A, 1_A, V\}$
6	p_{R2}	cmd_{R2}	$\{2_B, 2_B, V'\}$	$\{2_B, 1_A, V\}$	$\{1_A, 1_A, V\}$
7	p_{R3}	cmd_{R3}	$\{2_B, 2_B, V'\}$	$\{2_B, 1_A, V\}$	$\{1_A, 1_A, V\}$

Example 8.3: Concurrent read and write commands.

In contrast, all three read commands are blocked by the partially executed write. Commands cmd_{R1} and cmd_{R2} receive inconsistent read rounds and thus proceed to the read phase, whereas the proposer of cmd_{R3} has to start a WriteThrough due to inconsistent write rounds. However, is this additional work necessary to ensure strong consistency?

The proposer of cmd_{R1}, denoted as p_{R1}, can safely return a result based on V if it can prove that cmd_W was not chosen at the time it received cmd_{R1}. Otherwise, property P2 would be violated. For that, the proposer has two options: (1) show that cmd_W was chosen after p_{R1} received replies from a k-quorum, or (2) show that cmd_W was submitted concurrently to cmd_{R1}.

(1) can not be proved based on the information p_{R1} received. In fact, it can be contradicted by a slight modification of the example. Assume that a_3 has also received the write message in step 4. Then, cmd_W would have been chosen and p_{R1} still receives the same information from its k-quorum.

However, p_{R1} can prove that (2) applies, because it received a k-quorum with consistent write rounds. This means that all acceptors in this k-quorum received the message from p_{R1} before they voted for cmd_W. Since p_{R1} must have received command cmd_{R1} before sending the messages to the k-acceptors, only a k-minority could have voted for cmd_W before cmd_{R1} was submitted. Thus, cmd_W was not chosen before the read command started. Conversely, cmd_{R1} does not happen before cmd_W because the proposer received a reply with an incremented read round. Thus, cmd_W and cmd_{R1} are concurrent.

Note that examining the write rounds is enough to ensure that cmd_W was not chosen before cmd_{R1} was submitted, which is sufficient for p_{R1} to return the read command's result. Furthermore, the only information necessary to reach this conclusion was that the proposer received consistent write rounds. No assumptions specific to the example were made. Therefore, this argument applies in general. This leads to the following modification of the protocol:

Change 2. *A proposer receiving consistent write rounds can always return the result of a read command.*

The proposer of cmd_{R2} also receives consistent write rounds, therefore it can return the result of the command as well.

Change 2 does not need to know of any interfering or commuting relationship between the respective commands. However, this changes when the proposer of the read command receives inconsistent write rounds.

The proposer p_{R3} of cmd_{R3} in Example 8.3, receives inconsistent write rounds. p_{R3} can deliver the command's result depending on whether $cmd_W \parallel cmd_{R3}$ holds or not. Let rf_x denote the ReadFilter of cmd_x.

First, assume that $cmd_W \nparallel cmd_{R3}$. This means $cmd_W \asymp cmd_{R3}$, because $cmd_R \not\asymp cmd_W \forall cmd_R \in Reads$. Therefore, $rf_{R3}(V') \neq rf_{R3}(V)$. Assuming p_{R3} decides to return the result of cmd_{R3} with the k-quorum it received, it must decide which of the two values can be returned safely.

In the provided example, p_{R3} can not return $rf_{R3}(V)$, because it is possible that cmd_W was chosen before cmd_{R3} was submitted. p_{R3} has no information on acceptor a_3's state. It is possible that a_3 has voted for cmd_W in step 4, which means cmd_W could be chosen already. Additionally, p_{R3} can also not return $rf_{R3}(V')$, because it is still not certain that cmd_W will be the next chosen command. Another write command might receive votes from a_2 and a_3 if cmd_W's remaining write messages are delayed. Therefore, p_{R3} can not return either value safely. It has no other option than helping to establish cmd_W in a WriteThrough.

Now, assume $cmd_W \parallel cmd_{R3}$. This means $rf_{R3}(V') = rf_{R3}(V)$. Therefore, it does not matter if cmd_W was chosen or not for the result of cmd_{R3}. Thus, p_{R3} can return the result of the read command. In this example, it is clear that no intermediate write command was chosen before cmd_W because the write round of a_1 is only incremented by one compared to a_2's write round. However, this is not necessarily the case in general because any number of write commands can be proposed concurrently. An intermediate write command might not commute with cmd_{R3}. In this case, $rf_{R3}(V') \neq rf_{R3}(V)$, thus preventing the proposer to deliver the read. Therefore, a more generalized argument must be made:

Assume a client submits command cmd_R with ReadFilter rf_R to proposer p. The proposer executes the round_request phase and receives replies from a k-quorum Q of size n. The highest write round received in these messages is denoted as r_{h*} and the lowest round as r_{l*}. Let cmd_x be the write command voted by an k-acceptor in round r_x with v_x as the acceptor's value. p can receive multiple replies with the same write round. However, remember that cmd_x and v_x is consistent for all r_x across k-acceptors due to property $P3$ of PRBR. Assume that p receives inconsistent write rounds, thus $r_{h*} > 0$. This implies that there exists a command cmd_{prev} in write round $r_{prev} \geq 0$ that is the predecessor of cmd_{h*}. Figure 8.1 servers as an overview of the notation. Note that p might not necessarily receive a reply with write round r_{prev}. In this case, it is possible that $r_{prev} < r_{l*}$.

Proposition 1. *If $cmd_{h*} \parallel cmd_R$, then p can return $rf_R(v_{h*})$ without violating strong consistency.*

Proof.

1. $cmd_{h*} \parallel cmd_R$ implies $cmd_{h*} \not\asymp cmd_R$. Since cmd_{prev} is defined to be the predecessor of cmd_{h*}:

$$rf_R(v_{prev}) = rf_R(v_{h*}). \tag{8.3}$$

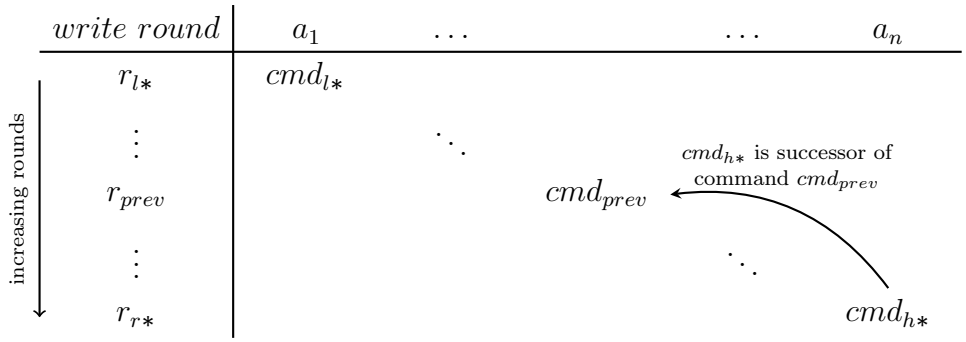


Figure 8.1: Responses to p form a k-quorum Q of size n sorted by write rounds.

2. cmd_{prev} is chosen in round r_{prev}.

   Assume that cmd_{prev} is not chosen. The proposer p_{h*} trying to establish cmd_{h*} can receive consistent or inconsistent write rounds in the round_request or read phase.

   (a) p_{h*} receives consistent write rounds. The received write round can not be equal to r_{prev}, since cmd_{prev} only exists in a k-minority. Thus, cmd_{h*} is proposed as successor of a different command. However, this contradicts the original assumption that cmd_{prev} is cmd_{h*}'s predecessor.

   (b) p_{h*} receives inconsistent write rounds. Only WriteThrough commands can be proposed after receiving inconsistent write rounds. However, WriteThroughs do not commute with any other command. Thus, this contradicts $cmd_{h*} \parallel cmd_R$.

   Both possibilities lead to contradictions. Therefore, cmd_{prev} is chosen.

3. There does not exist a command other than cmd_{h*} which is chosen in a round higher than r_{prev}.

   (a) No command is chosen in round r_h with $r_{prev} < r_h < r_{h*}$.

   Assume a chosen command cmd_h exists. cmd_h can be either chosen before, or after a_n voted for cmd_{h*}.

      i. cmd_h was chosen before a_n's vote.
      Before a_n could have voted for cmd_{h*}, some proposer p' must have received consistent write rounds from some k-quorum Q'. At this moment, all acceptors in Q' have read round $\geq r_{h*}$, thus only a k-minority can still vote for cmd_h. Since p' received consistent write rounds, either all or no k-acceptors in Q' have voted for cmd_h. If no k-acceptor has voted for cmd_h, then cmd_h can not be chosen anymore because only a k-minority can still vote for a command in a round smaller than r_{h*}. If all k-acceptors voted for cmd_h, then cmd_{h*} is proposed as a successor of cmd_h. However, $cmd_{prev} \neq cmd_h$ because $r_{prev} < r_h$. Therefore, this creates a contradiction.

      ii. cmd_h was chosen after a_n's vote.

This is not possible because at most a k-minority can vote in a write round smaller than r_{h*} after a_n's vote for cmd_{h*}, as established in 3(a)i.

Therefore, no chosen command cmd_h can exist.

(b) No command is chosen in round r_h with $r_h > r_{h*}$.

Assume a chosen command cmd_h exists. Since r_{h*} denotes the highest received write round, Q does not contain an acceptor which has voted for cmd_h. Therefore, at most the remaining k-minority has voted for cmd_h. This contradicts the assumption that cmd_h is chosen.

(c) No k-acceptor votes for a command other than cmd_{h*} in round r_{h*}.

As per protocol, every command is proposed with a unique round ID. Therefore, there can not be two commands with the same round.

Note that cmd_{h*} might or might not be chosen. Based on the information received by p, a_n could be the only k-acceptor which has voted for cmd_{h*}, or every acceptor in the not visible k-minority might have voted for cmd_{h*}.

4. Every read command $cmd_{R'}$ with ReadFilter $rf_{R'}$ returns a result that is equivalent to some value $rf_{R'}(v_x)$, where r_x is a round in which a write command was chosen.

   A proposer p' can return the result of a read command if one out of two conditions apply: Either p' received consistent write rounds as per Change 2, or the command voted for in the highest received write round, denoted as $cmd_{h*'}$, commutes with $cmd_{R'}$, as per Proposition 1.

   (a) p' received consistent write rounds.

   Some k-quorum exists that has voted for a command in this round. Therefore, a command is chosen in this round.

   (b) p' receives a k-quorum with $cmd_{h*'} \parallel cmd_{R'}$.

   Let $r_{prev'}$ be the round in which the predecessor command of $cmd_{h*'}$ was voted for. Due to step 1, $rf'_R(v_{h*'}) = rf'_R(v_{prev'})$. Because of step 2, $cmd_{prev'}$ was chosen in round $r_{prev'}$

5. Step 2 and 3 of the proof show that the most recently chosen command is either cmd_{prev} or cmd_{h*}. Either way, the result of cmd_R is the same because of step 1. Therefore, cmd_R can not return an outdated value.

   Furthermore, step 4 shows that all other reads returned a value equivalent to a value with cmd_{prev}, cmd_{h*}, or some older command cmd_l as the most recent command in the command sequence. In case of cmd_l or cmd_{prev}, cmd_R returns a newer value because its ReadFilter is applied on cmd_{h*}. If cmd_{h*} is the case, then both commands applied their ReadFilter on the same state of the command sequence. Therefore, there does not exist a read which has returned a newer value than cmd_R.

   Thus, p can return $rf_R(v_{h*})$ without violating strong consistency.

□

The implication of Proposition 1 is, that only the reply with the highest write round must be examined to decide whether the result of a read command can be delivered or not. This leads to the following modification of PRBR:

Change 3. *A proposer can return the result of a read command cmd_R, if the write command, which was voted for in the highest received write round, commutes with cmd_R.*

8.3.2 Modifying PRBR

Change 3 requires knowledge of the command that every acceptor has voted for. In PRBR's current state, an acceptor receiving a write command with valid rounds in the write phase of the protocol will apply the command to its current value. However, it does not remember the write command.

Therefore, the state held by acceptors must be extended by a state variable cmd_{write}, which represents the last command the respective acceptor has voted for. In PRBR's write phase, proposers must now send the full command to the k-acceptors. If a k-acceptor receives the command with valid rounds, then it updates cmd_{write} along with its remaining state variables (the read round, write round, and value held by it). These changes are depicted in Code Listing 8.3 in line 26 and 33, respectively.

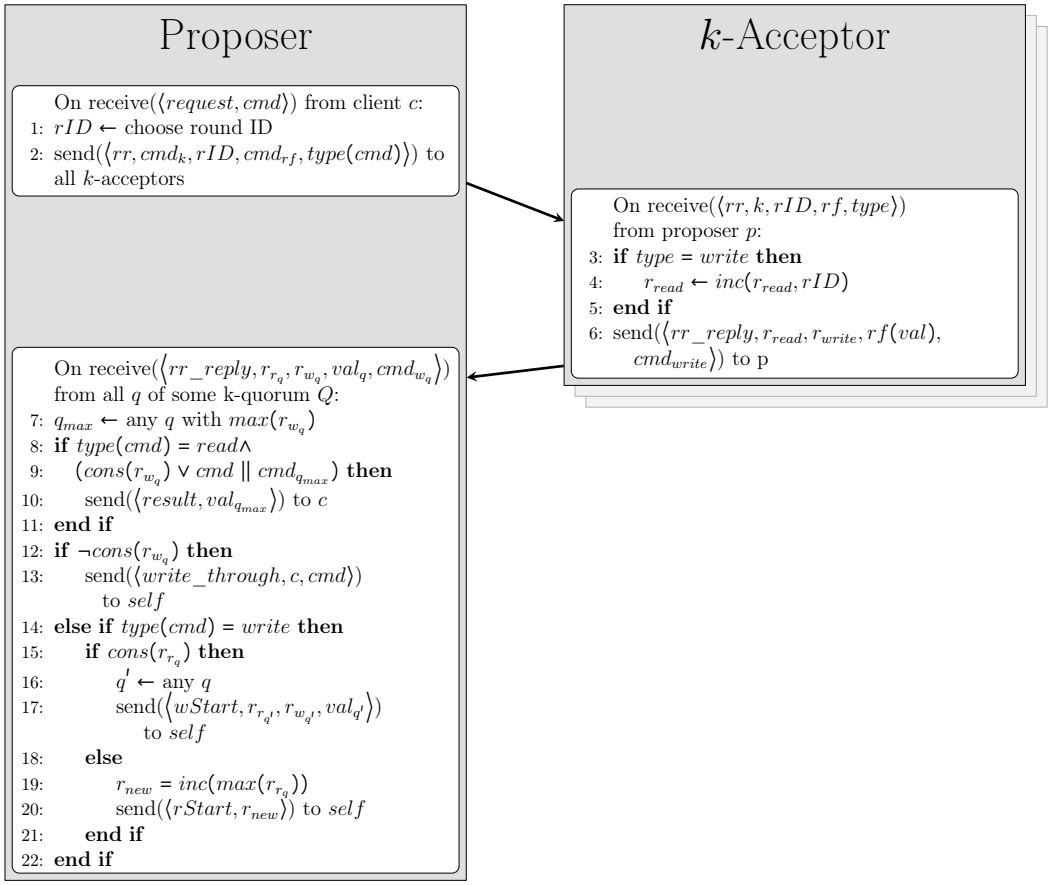
k-acceptors can now include the last command they voted for in their *rr_reply* messages. Due to Change 2 and Change 3, the condition in which proposers can return the result of read commands changes. It can be returned if either the received write rounds are consistent (Change 2) or if the last write command, that the k-acceptor in the seen k-quorum with the highest write round voted for, commutes with the current read command (Change 3). This modification is depicted in Code Listing 8.2 line 8-9.

Note that proposers processing read commands never enter PRBR's read phase. Previously, the read phase was entered if the proposer received inconsistent read rounds. As per Change 2, this is not necessary to satisfy strong consistency. However, it is still possible for such a proposer to start a WriteThrough if the two conditions above are not satisfied.

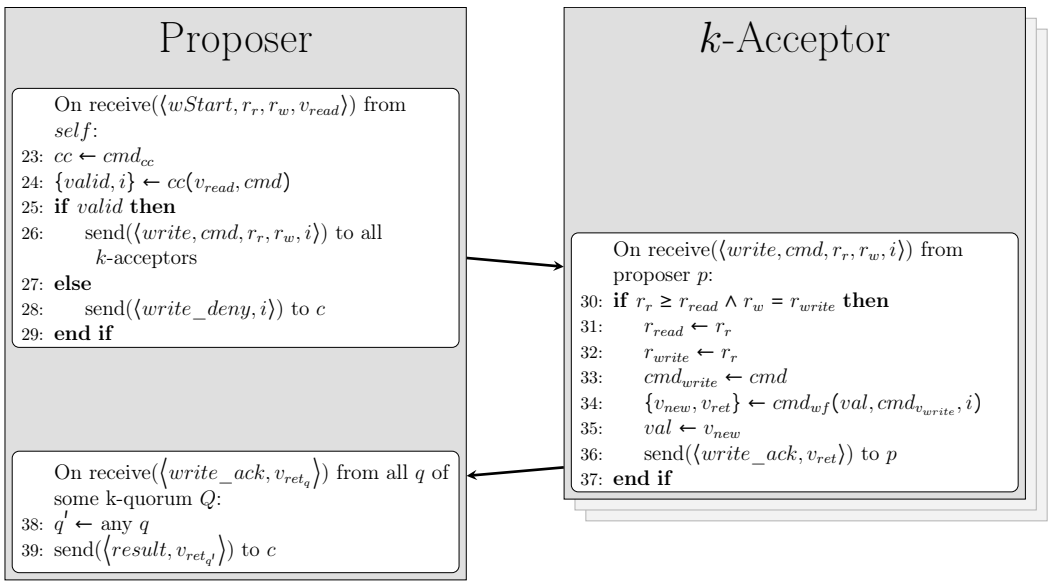
The logic concerning write commands remains unchanged. If a proposer receives inconsistent write rounds when processing a write command, it starts a WriteThrough. If it receives consistent write rounds but inconsistent read rounds, it proceeds to the read phase. If both rounds are consistent, then it can start the write phase of PRBR.

8.3.3 Impact and Further Optimizations

The modifications made to PRBR in this section increase the size of some messages send as part of the protocol. First, k-acceptors send the last write command they have voted for as an additional element in their *rr_reply* messages. Second, *write* messages must include the full write command. Parts of the command were already included in this message. Thus, only the ContentCheck and ReadFilter of the command must be transmitted in addition. Furthermore, the size of each acceptor's state increased because they have to store the last command they have voted for.



Code Listing 8.2: Modifications in PRBR's round_request phase to support reads commuting with concurrent writes.



Code Listing 8.3: Modifications in PRBR's write phase to support reads commuting with concurrent writes.

The size of write commands can vary. The included write value has the most profound effect on the command's size since this value can be arbitrarily large. A write command writing binaries that are multiple megabytes large will also cause command sizes of multiple megabytes. This would cause very high overhead in the round_request phase of the protocol.

However, the modifications as described in the previous section are pessimistic. Usually, the decision whether a read command commutes with a write command does not depend on the write value of the command, but instead on the operation performed by the write command on the stored data, i.e. the command's filters. In this case, only the filters of the write commands must be stored and transmitted. The size of a filter is more predictable. For languages like Erlang, only sending a reference to the function defining the filter is necessary. This requires only a small, relatively constant amount of data.

The overhead can be reduced further if the set of possible write commands is fixed and known beforehand. In this case, each possible filter triple can be assigned to a command ID. Then, only this ID has to be stored and included in messages. Since the command ID is enough to identify the filters of a command, it is sufficient to decide if a read command commutes with a write command with a given ID. Naturally, the size of the command ID depends on the number of possible commands. However, it only grows logarithmic in size and is therefore typically small.

When processing read commands that access an item concurrently to a write command, the modifications made in this section increase the likelihood that reads can be delivered immediately without additional work. Therefore, the number of messages sent and message delays needed to process read commands decreases on average. Since no read commands enter the read phase, reads do not block writes from succeeding anymore. However, read commands can still trigger WriteThroughs, thus bringing the potential of causing dueling proposers.

In all other cases, the flow of the protocol remains unchanged.

8.4 Commutative Writes

This section extends PRBR to prevent commuting, concurrent write commands from blocking each other. As with the previous sections, it first must be established what it exactly means for two write commands to commute.

A write command can be roughly split into two parts. The first part of a write command is the validation part that checks whether the command can be applied to the current value of the item stored in the k-acceptors. This includes the ReadFilter, that reads the item's current value (or part of it), and the ContentCheck performing the actual validation. In the time between application of the ReadFilter on acceptors and ContentCheck, no other write command can interfere with the result of the validation. If, however, two write commands cmd_1 and cmd_2 commute, then the proposer of cmd_2 might receive some responses of acceptors that already have applied cmd_1 and some others without cmd_1. If cmd_2 commutes with cmd_1 this state should be allowed because otherwise cmd_1 blocks cmd_2. It is important that cmd_2 passes its ContentCheck with or without cmd_1 being applied to replicas. Otherwise the result of cmd_2 would depend on cmd_1, which contradicts the definition of commuting commands.

The second part of a write command consists of the WriteFilter, the write value submitted by the client and the UpdateInformation returned as part of the result of the command's validation. This information is enough to express the write operation performed by the command. For two write commands to commute, the value of the item must be the same after application of the commands in either order. Otherwise, the value held by acceptors would be inconsistent after both commands have completed.

After a completed write, the respective proposer collects *write_ack* messages containing the return value for the client. Since it should be possible to apply commuting commands in either order, the proposer of cmd_2 can also receive replies from k-acceptors that might or might not have applied cmd_1. At this stage, the proposer of cmd_2 should not be forced to differentiate between replies and try to find out whether cmd_1 was chosen or not. Therefore, the return result of the command's WriteFilter should be unaffected by the existence of the other command. By doing so, the client will receive the same result either way.

This condition might sound hard to fulfill. However, it is sufficient in many scenarios (for example all write commands used as part of Scalaris transaction protocol) that write commands simply return a fixed "acknowledge" value that indicates that the write was successful. Since the returned value is fixed, the order in which commuting commands are applied does not matter.

Some simple examples for write commands that can be modeled this way include: incrementing a value, or adding/removing elements from a set. The later can be used as read lock acquisition as part of a transaction protocol, since the order in which read locks are acquired generally does not matter. For such a command, the respective ContentCheck might test for the existence of a write lock. The ContentCheck validates a different part of the value as is modified by the read lock command. Therefore, it is easy to see that previous instances of the command have no impact on a successful validation.

As with reads commuting with writes, it is assumed that some mechanism exists to identify which write commands commute with each other.

8.4.1 Identifying Avoidable Conflicts

Analogously to the previous sections, an example helps to identify were conflicts between two concurrent write commands can occur. Example 8.4 depicts the first phase of the protocol when handling concurrent writes.

step	*prop.*	*cmd*	a_1	a_2	a_3
1	–	–	$\{1_A, 1_A, V\}$	$\{1_A, 1_A, V\}$	$\{1_A, 1_A, V\}$
2	p_{W1}	cmd_{W1}	$\{2_B, 1_A, V\}$	$\{2_B, 1_A, V\}$	$\{1_A, 1_A, V\}$
3	p_{W2}	cmd_{W2}	$\{2_B, 1_A, V\}$	$\{3_C, 1_A, V\}$	$\{2_C, 1_A, V\}$
⋮				⋮	

Example 8.4: Start of two concurrent, conflicting write commands

The proposer of write command cmd_{W1} has completed the round_request phase of PRBR. After that, the proposer of cmd_{W2} received replies from a different

k-quorum, therefore receiving inconsistent read rounds. When continuing the example, p_{W2} will start the read phase and hopes to receive promises from a k-quorum in round 4. Furthermore, at most one acceptor (a_1) can accept the write command sent by p_{W1}. p_{W1} will therefore retry its write in the future.

Both commands block each other from succeeding. If $cmd_{W1} \nparallel cmd_{W2}$, then this can not be avoided because a proposer needs a k-quorum of promises before it can safely propose its command. If, however, $cmd_{W1} \parallel cmd_{W2}$, then it should be possible to process both commands on all acceptors in arbitrary order without creating conflicts.

When comparing Example 8.4 with the example of conflicting read commands, depicted in Example 8.1, strong similarities between the two problems can be seen. The latter was solved by not incrementing read rounds. This allowed concurrent reads to receive consistent read rounds. Effectively, this made reads invisible to other reads because no information that a read occurred was persisted in acceptors. Since all reads commute with each other, this did not cause violations of strong consistency.

Of course, writes do not commute with all other writes. A write command must know the existence of a concurrent, non-commuting write, but not necessarily of a commuting write. By extending the idea of commuting reads, sets of commuting writes can be executed in the same round until a non-commuting write is submitted.

8.4.2 Command Sets

The commutative properties of write commands are modeled by assigning write commands to so-called command classes.

Definition 12 (Command Class)**.** *A command class C is a predefined set of write commands that commute with each other:*

$$cmd_1 \parallel cmd_2 \qquad \forall cmd_1, cmd_2 \in C \tag{8.4}$$

Each command class is required to be disjunct from all other command classes. This makes it easier to decide whether two write commands commute during runtime. If their class matches, they are considered to commute, otherwise not. Note that by doing so, it is only possible to model commutative relationships that are symmetric and transitive. This restriction is a trade-off for a simpler protocol design.

A write command that does not commute with any other command (including itself) is not assigned a command class. If required, the null value $\perp$ will be used to denote the absence of a class. All read commands are considered to be of class *read*.

Up until now, PRBR established a sequence of commands for each item. In each round, k-acceptors could vote for at most one write command. Now, this command sequence will be extended to a sequence of command sets.

Definition 13 (Command Set)**.** *A command set is a set of write commands with the same command class, that are voted for in rounds with the same round number.*

Hereinafter, command sets will be abbreviated as c-sets. The class of a c-set is the class that all write commands in it share. A write command commutes with a c-set if it commutes with all of the c-set's write commands, i.e., if the write command is of the same class as the c-set.

8.4.3 Sequence of Command Sets

Unlike previous changes of PRBR, extending the protocol to use command sets can not be done by locally limited modifications. Instead, every phase of the protocol must be changed. All changes introduced in this section are connected and thus can not be implemented in isolation.

Furthermore, the current modifications of PRBR represent only a draft. Their complete correctness is not proven unlike the changes made in the previous sections. Due to the number of changes required, the level of detail in the argumentation of each change is reduced compared to the previous sections.

Choosing the Round of the next Command

Section 8.4.1 introduced the first issue that has to be solved when extending PRBR to c-sets: It must be possible for commutative write commands to proceed in the same round, whereas a non-commutative command must use a different round.

This problem is reminiscent of the scenario with two concurrent reads. Thus, the solution is similar as well.

Change 4. *An acceptor receiving a write command in the round_request phase does not increment its read round, if the previous write command it has received in the round_request or read phase had the same command class.*

It is not enough for an acceptor to remember the class of the command it has voted for last. In this case, two concurrent, commutative commands that enter the round_request phase before acceptors have voted for either command would still block each other.

An example illustrates how Change 4 affects the flow of PRBR's first phase.

step	*prop.*	*cmd*	*class*	a_1	a_2	a_3
1	–	–	$\perp$	$\{1_A, 1_A, V\}$	$\{1_A, 1_A, V\}$	$\{1_A, 1_A, V\}$
2	p_{W1}	cmd_{W1}	c_1	$\{2_B, 1_A, V\}$	$\{2_B, 1_A, V\}$	$\{1_A, 1_A, V\}$
3	p_{W2}	cmd_{W2}	c_1	$\{2_B, 1_A, V\}$	$\{2_B, 1_A, V\}$	$\{2_C, 1_A, V\}$
4	p_{W3}	cmd_{W3}	c_2	$\{2_B, 1_A, V\}$	$\{3_D, 1_A, V\}$	$\{3_D, 1_A, V\}$
⋮					⋮	

Example 8.5: First phase of write commands making use of command classes.

cmd_{W1} and cmd_{W2} have the same command class, therefore they commute with each other. The proposers of the commands receive responses from different k-quorums. Due to Change 4, they receive replies with read round number 2. Note that the round ID of a_3 differs from a_1 and a_2 because a_3 has received cmd_{W2} first. This is unavoidable without global knowledge of all requests submitted to all proposers or a distinguished leader process. Therefore, two write commands with the same class only have to be proposed in the same round number to be

considered part of the same c-set. For commands with differing classes, the whole round will be used for comparisons as usual.

Command cmd_{W3} does not commute with the other two commands. This causes acceptors to increment their read rounds. Thus, at most a_1 can vote for cmd_{W1} or cmd_{W2}, which means that they can not be chosen in round 2. This reflects the original behavior of PRBR.

Examining the Received Replies

Before a proposer p can deliver the result of a read command or propose a write command in PRBR's write_phase, it must examine the replies it received from the k-quorum and make a decision based on this information. This decision-making process must be extended to c-sets.

(1) First, assume that p has received a read command cmd_R. Previously, p was able to deliver the result of cmd_R if one of two conditions were satisfied: (a) p received consistent write rounds, or (b), cmd_R commutes with the command voted for in the highest received write round.

Acceptors can vote for an arbitrary number of commands in the same round if they have the same class. If cmd_R is proposed concurrently to write commands, acceptors in the k-quorum might have voted for different sets of commands since not all acceptors might have received all messages yet. However, this can not be deducted by p solely by examining the received write rounds. p must have knowledge of the most recent c-set of each k-acceptor.

step	*prop.*	*cmd*	*class*	a_1	a_2	a_3
1	p_{W1}	cmd_{W1}	c_1	$\{1_A, 1_A, [A, B]\}$	$\{1_A, 0, []\}$	$\{1_A, 1_A, [A]\}$
2	p_{W2}	cmd_{W2}	c_1	$\{1_A, 1_A, [A, B]\}$	$\{1_A, 1_A, [B]\}$	$\{1_A, 1_A, [A]\}$
3	p_{W1}	cmd_{W1}	c_1	$\{1_A, 1_A, [A, B]\}$	$\{1_A, 1_A, [B, A]\}$	$\{1_A, 1_A, [A]\}$
4	p_{R1}	cmd_{R1}	*read*	$\{1_A, 1_A, [A, B]\}$	$\{1_A, 1_A, [B, A]\}$	$\{1_A, 1_A, [A]\}$
5	p_{R2}	cmd_{R2}	*read*	$\{1_A, 1_A, [A, B]\}$	$\{1_A, 1_A, [B, A]\}$	$\{1_A, 1_A, [A]\}$

Commutative write commands cmd_{W1} and cmd_{W2} each add an element to a set. The c-set of a_1 and a_2 contains both write commands, whereas a_3's c-set contains only cmd_{W1}. Both proposers p_{R1} and p_{R2} receive consistent rounds. However, p_{R2} cannot deliver the read since it does not know if cmd_{W2} is chosen or not.

Example 8.6: Example of proposers receiving inconsistent c-sets.

In addition to comparing write rounds, p can then compare the received c-sets for equality. If they are consistent, p can deduce that all commands it has seen are chosen, and all other commands voted for in this round exist in at most a k-minority. p does not need to care about the order the acceptors have voted for the commands because by the definition of commutative commands, the resulting value of the item must be the same for all permutations. This covers case (a). For simplicity, a *consistent write state* refers to receiving consistent write rounds and consistent c-sets in a k-quorum.

Change 5. *A proposer can deliver the result of a read command, if it has received a consistent write state.*

Condition (b) is the result of proving Proposition 1. An intermediate result of the proof was, that the last chosen command in the command sequence is either some (possibly unknown) command cmd_{prev} or the command voted for in the highest received round, cmd_{h*}. If $cmd_R \parallel cmd_{h*}$, then the result of cmd_R is the same either way, thus p can return the value received in the reply with the highest round number.

By extending this to c-sets, p might receive multiple commands which were voted for in the highest received round r_{h*}. To ensure that the delivery of cmd_R's result does not violate strong consistency, cmd_R must now commute with *all* write commands for which p can not be certain if they are chosen or not. These are all commands received in r_{h*} which were not included in all replies of the k-quorum. For example, read command cmd_{R2} of the example depicted in Example 8.6 can return if $cmd_{R2} || cmd_{W2}$. Of course, if p received inconsistent write rounds, no command received in r_{h*} is included in every reply. In this case, cmd_R must commute with all those commands.

Change 6. *A proposer can return the result of a read command cmd_R, if all write commands, which were voted for in the highest received write round but are not for certain chosen, commute with cmd_R.*

If the conditions of Change 5 and Change 6 are not fulfilled, then p cannot deliver the result of cmd_R and must start a WriteThrough.

(2) Next, assume that p has received a write command cmd_W. As with reads, there are now two conditions for which p can continue to the write phase: (a) p received a consistent quorum, or (b) cmd_W can be proposed as part of the c-set with the current highest round.

Condition (a) is unchanged to unmodified PRBR. However, its semantic is slightly different. Before, a consistent quorum required consistent read and write rounds. A proposer receiving both could be sure that no command was proposed in a round higher than the received write round. Examining write rounds is not enough anymore, because k-acceptor might have voted for different c-sets. Therefore, a consistent quorum now requires consistent read rounds and a consistent write state. If the received read round is higher than the write round, then p will start a new c-set by proposing cmd_W.

Condition (b) implies that cmd_W commutes with all commands received as part of the most recent c-set. By definition, this is the case when cmd_W's class matches with the class of the commands in the c-set. In addition, p must be certain that no newer c-set was started yet. If the highest received read round matches with the write round of the most recent c-set, then no non-commuting write command could have received a k-quorum of round_request replies for a higher round and thus no k-acceptor could have voted for any command with a higher round (see Example 8.7 for an example).

Change 7. *A proposer can propose a write command in PRBR write phase, if it can be proposed as part of the highest seen c-set.*

If neither (a) nor (b) applies, then p must either start the read_phase if it has received inconsistent read rounds to receive a k-quorum of promises or must start a WriteThrough if it has received an inconsistent write state. This remains unchanged to unmodified PRBR.

step	*prop.*	*cmd*	*class*	a_1	a_2	a_3
1	p_{W1}	cmd_{W1}	c_1	$\{1_A, 1_A, [A]\}$	$\{1_A, 1_A, [A]\}$	$\{1_A, 1_A, [A]\}$
2	p_{W2}	cmd_{W2}	c_2	$\{2_B, 1_A, [A]\}$	$\{1_A, 1_A, [A]\}$	$\{1_A, 1_A, [A]\}$
3	p_{W3}	cmd_{W3}	c_1	$\{3_C, 1_A, [A]\}$	$\{1_A, 1_A, [A]\}$	$\{1_A, 1_A, [A]\}$
4	p_{W4}	cmd_{W4}	c_1	$\{3_C, 1_A, [A]\}$	$\{1_A, 1_A, [A]\}$	$\{1_A, 1_A, [A]\}$

All acceptors vote for write command cmd_{W1}. a_1 receives a round_request message for command cmd_{W2} with a different class and increments its read round. This causes p_{W3} to receive inconsistent read rounds. From p_{W3}'s knowledge, it is possible for a_3 to have voted for cmd_{W2} in round 2. Furthermore, a_2 can still vote for cmd_{W2}, making cmd_{W2} a chosen command. p_{W3} can not proceed and must try to receive a k-quorum of promises in a higher round. In contrast, p_{W4} knows that no k-acceptor could have voted for a command in a higher round. Therefore, it can proceed to the write phase in round 1.

Example 8.7: Example of proposers receiving inconsistent read rounds.

Voting for Write Commands

If proposer p is trying to establish write command cmd_W and the k-quorum of replies it received satisfied one of the conditions outlined in the previous section, then p proceeds to PRBR's write phase as usual. Once the ContentCheck of cmd_W passes, p sends write messages to all k-acceptor to propose cmd_W.

In the original version of PRBR, it was necessary for a proposed command to satisfy two conditions before an k-acceptor a was able to vote for it: the command must have been proposed in a higher or equal round than the highest round a has given a promise for, and the last command a voted for must have been the last chosen command (see line 8 in PRBR's write phase on page 34).

The general nature of these conditions remains unchanged if a has received a non-commuting write command, i.e. the write command can not be included in a's current c-set. This is analogous to voting for a new command in PRBR's original version. As shown in the example in Example 8.9, it is no longer enough to just examine the write round, since it is possible for an acceptor to vote for more than one command in a round. Therefore, p must include all chosen commands of the previous write round as a c-set in its write message so that a can check that it has voted for the same set of commands in the previous round.

Change 8. *Acceptor a can vote for a command cmd_W in a higher round (and thus starting a new c-set) if a has not given a promise in a higher round and has voted for exactly the set of commands that were chosen as part of the previous c-set.*

If cmd_W commutes with the current c-set of a, then a can vote for cmd_W if it can be included in the c-set. By the definition of c-sets, this requires cmd_W to be proposed in the same round as all other commands of this set. Naturally, a must not have given a promise in a higher round as well, since that would violate one of the underlying principles of Paxos. Example 8.7 can be revisited for an example of adding commands to an existing c-set.

Change 9. *Acceptor a can vote for command cmd_W, if a has not given a promise in a higher round, cmd_W commutes with a's current c-set and is proposed in the same round.*

step	*prop.*	*cmd*	*class*	a_1	a_2	a_3
1	p_{W1}	cmd_{W1}	c_1	$\{1_A, 1_A, [A, B]\}$	$\{1_A, 1_A, [A]\}$	$\{1_A, 1_A, [A]\}$
2	p_{W2}	cmd_{W2}	c_2	$\{2_B, 1_A, [A, B]\}$	$\{2_B, 1_A, [A]\}$	$\{2_B, 1_A, [A]\}$
3	p_{W2}	cmd_{W2}	c_2	$\{2_B, 1_A, [A, B]\}$	$\{2_B, 2_B, 1\}$	$\{2_B, 2_B, 1\}$

Write commands of class c_1 add values to a set. Acceptor a_1 has voted for a command that was not chosen yet. p_{W2} receives a consistent quorum and can therefore propose the command. Here, cmd_{W2} simply replaces the set of values by the size of the set. a_1 is not allowed to vote for the command. A WriteThrough is necessary to synchronize a_1 with the other acceptors.

Example 8.8: Example of proposer starting a new c-set.

WriteThrough

The last part of the protocol that has to be extended is the WriteThrough mechanism. Its purpose is to synchronize the state of k-acceptors if they have voted for different interfering write commands. Previously, a proposer executing a WriteThrough could simply use the state of the acceptor with the highest received write round because there could not exist a chosen command in a higher round.

The extension of WriteThroughs to c-sets is straight forward. When a proposer p receives a k-quorum of replies in the round_request or read phase, it must ensure that no chosen command is discarded.

p receives the full value of each k-acceptor (due to ReadFilter of the WriteThrough), along with their most recent c-set. Based on this information, p can construct a value that is safe to write. It can choose one of the received replies with maximal write round and apply all missing write commands received as part of this c-set on it in arbitrary order. This is allowed because all commands commute, therefore their execution order does not matter. That way, some not yet chosen commands are included in the WriteThrough, but no chosen command is discarded (since at most a k-minority can have voted for any not received command).

Change 10. *WriteThroughs can write a value based on the union of all c-sets received in the highest round.*

Example 8.9 shows the progress of a possible WriteThrough.

step	*prop.*	*cmd*	*class*	a_1	a_2	a_3
1		–	–	$\{1_A, 1_A, [A, C]\}$	$\{1_A, 1_A, [B, A]\}$	$\{1_A, 1_A, [B, D]\}$
2	p_A	cmd_{WT}	$\perp$	$\{2_B, 1_A, [A, C]\}$	$\{2_B, 1_A, [B, A]\}$	$\{1_A, 1_A, [B, D]\}$
3	p_A	cmd_{WT}	$\perp$	$\{2_B, 2_B, [B, A, C]\}$	$\{2_B, 2_B, [B, A, C]\}$	$\{2_B, 2_B, [B, A, C]\}$

Some write commands have started adding elements A through D to a set as part of the same c-set in round 1_A. The proposer of the WriteThrough receives replies of acceptors that have voted for A,B and C. Since p_A does not know if A, or C were chosen, they have to be included in the WriteThrough. All commands commute with each other because they were members of the same c-set. Thus, the execution order in the WriteThrough does not matter. The vote for D is discarded. However, this command was not and cannot ever be chosen in round 1_A.

Example 8.9: Example of WriteThrough in combination with c-sets.

8.4.4 Modifying PRBR

Extending PRBR to c-sets introduces a great amount of complexity to the protocol. The changes are depicted in Code Listing 8.4 and 8.5. The modifications of the read phase are analogous to the round_request phase. For the sake of brevity, they are not shown here.

The state of acceptors must be extended by three variables. First, the class of the last write command which caused the acceptor to increment its read round, $class_{prev}$. This is needed to implement the conditional increment of the acceptor's read round (Change 5). Furthermore, each acceptor must have access to all commands it voted for in its current c-set, denoted as $cset_{cur}$. This replaces cmd_{write}, which was previously used to remember the last command. $cset_{cur}$ is needed to apply missing commands as part of a WriteThrough and it is needed by proposers to check for consistent quorums. Finally, $cset_{prev}$, which is the c-set the acceptor has voted for before starting $cset_{cur}$. This is required to prevent acceptors from skipping chosen commands in their command sequence.

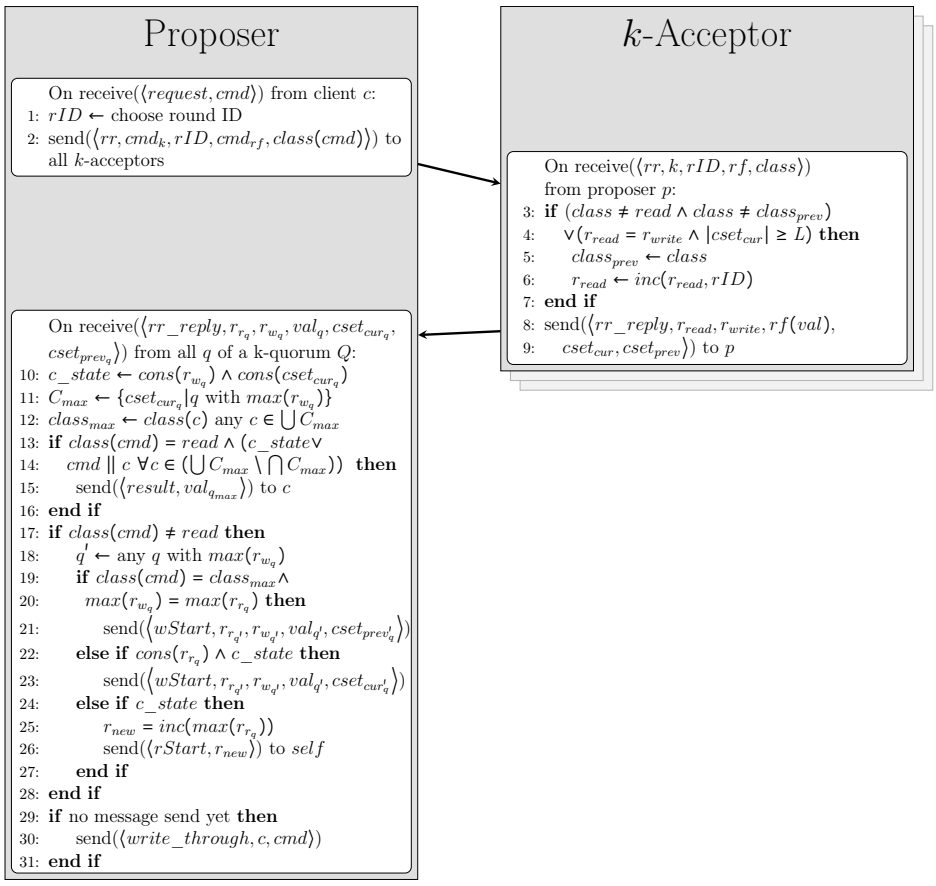
The protocol begins as usual with the proposer p choosing an ID and sending the ID, ReadFilter and class of the command to all k-acceptors (line 1). Each k-acceptor receiving the message now decides if its read round must be incremented. Line 3 encodes Change 1 and 4: an acceptor must only increment their read round if receiving a write command with a class that does not match the previous write command's class.

This causes an uninterrupted series of commutative write commands to be assegined to the same round, which in turn allows their proposer to propose them as part of the same c-set. However, this might cause unbounded growth of the current c-set if all proposed write commands commute with each other, for example if a key is used as a counter and only increment commands are proposed. This is problematic because growing c-sets cause the size of messages to increase, as well as the state held by acceptors. A limit on the possible size of c-sets is needed. A simple way to prevent unlimited growth of the c-set is to check the current c-set's size. If it has reached a constant threshold L, the acceptor will always increment its read round number (if it was not already incremented before). This is shown in line 4.

Eventually, a k-quorum has replied to the proposer's message. The responses contain the usual rounds and the (partial) value. In addition, acceptors must include their last two c-sets, $cset_{cur}$ and $cset_{prev}$, as well.

Based on the received information, p makes a decision about the next step in the protocol. This concerns Changes 5-7 discussed in the previous section. Most of the code of this action (line 10-31) is a straight-forward implementation of these changes.

The delivery conditions for reads – receiving a consistent write state (Change 5) or commuting with all commands which are not for certain chosen (Change 6) – are depicted in line 13 and 14, respectively. The original behavior of writes is shown from line 22 to 28: Either p receives a consistent quorum and can start the write phase of PRBR, or it receives a consistent write state but inconsistent read rounds and must continue to the read phase. Due to the extension to c-sets, p can now also propose a write command if it can be proposed as a member of the current c-set (Change 7), as is shown in line 20. Note that p must now include



Code Listing 8.4: Modifications in PRBR's round_request phase to support commutative write commands.

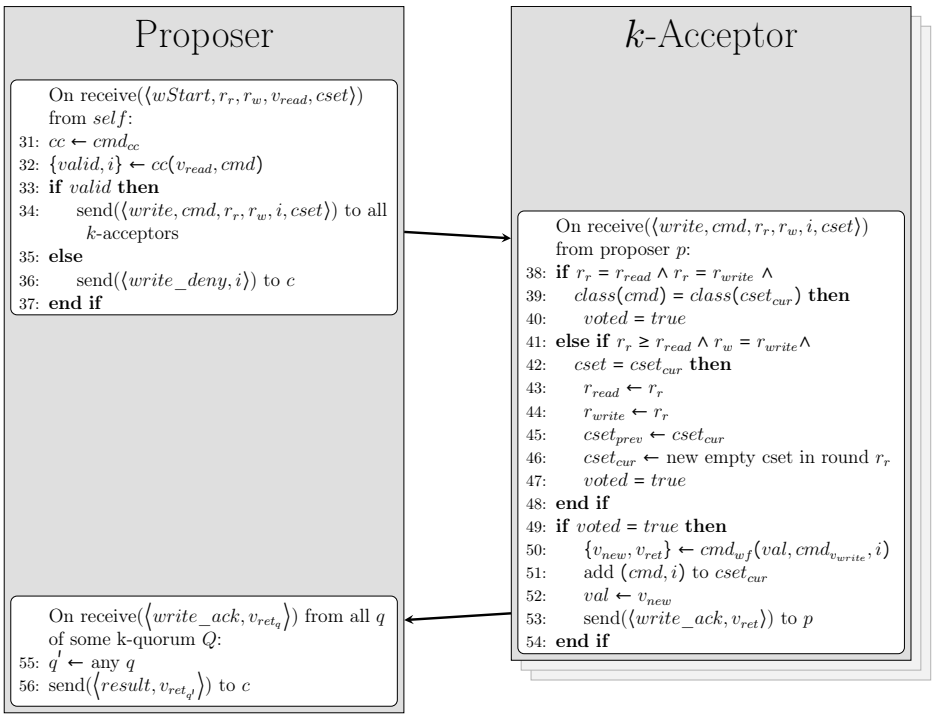
the predecessing c-set of the write command it will propose when continuing to the write phase. If the command will be proposed as part of the same c-set (line 21), then the predecessor remains unchanged. If, however, the command will be proposed in a higher round (line 23) and therefore in a new c-set, then the currently highest c-set becomes the new predecessor.

A WriteThrough is performed if no success condition for either read or write commands is met (line 29). This concludes the modification of PRBR's round_request phase. As already mentioned, the read phase is modified in the same manner and is thereby omitted here.

The beginning of the write phase remains largely unchanged. p checks if the write command it wants to propose is a valid successor by using the command's ContentCheck. If it passes, p sends a *write* message to all k-acceptors, with the predecessing c-set as an additional element (line 34).

An acceptor a can now vote for a write command if it satisfies one of two conditions: First, the command is proposed in a higher round and its predecessing c-set matches the c-set a has voted for last (Change 8). This is equivalent to the vote condition of PRBR's unmodified version. In addition to updating its rounds as usual, a must update its references to its current and previous c-set, because the received command starts a new one. This is shown from line 41 to 48.

Second, a can now vote for a write command that is of the same class and is



Code Listing 8.5: Modifications in PRBR's write phase to support commutative write commands.

proposed in the same round as its current c-set (Change 9). The write command commutes with all commands in this c-set and thus can be included in it. Since the rounds are the same and no new c-set is introduced, no further work must be done (line 38-40).

If either condition is satisfied, a can apply the command's write filter, update its stored value and include the command in its current c-set (line 49-54). Afterwards, it sends the result of the command back to the proposer. Once the proposer has received a k-quorum of *write_ack* replies, it knows that the command is chosen and can therefore deliver the result to the client.

8.4.5 Impact and Further Optimizations

Extending PRBR to c-sets increases the amount of data each acceptor must persist, as well as message sizes throughout the protocol. Due to the introduction of the c-set size limit, the state can not grow arbitrarily large over time, which is important for PRBR's initial in-place property.

However, it is important to note that the limit, as it is described in the previous section, merely imposes a soft limit on the c-sets size. Bursts of commutative write commands might cause the acceptor to reply to a high number of messages before the acceptor has actually included a new command in the write phase. A hard limit could be implemented by introducing a "budget" for each round. Every time an acceptor sends a promise without incrementing its read round in the round_request phase, the budget is decremented. Once depleted, the acceptor increments its round. This restricts the total amount of *rr_reply* message that can be sent by each acceptor for a given round. As an additional benefit, write commands can be weighted to accommodate varying command sizes.

Furthermore, it is not necessary most of the time to include the full commands in messages. Full commands are only needed during a WriteThrough when the proposer forms the union of all write commands and applies all missing write commands on top of the full value received from acceptors. During normal execution, it is enough to attach IDs to commands and only include those IDs in the messages. This greatly reduced messages sizes, especially for larger commands. Since commands included in $cset_{prev}$ c-sets of the acceptors are not used in WriteThroughs, it is enough to just store command IDs in it. This reduced the size of the state held by each acceptor.

Nevertheless, storing a single full c-set for every k-acceptor introduces high additional storage overhead. The maximum size of c-sets must be carefully chosen depending on the command sizes and size of the actual values. If only a single integer is stored for every item, this overhead might be many times more than that of the actual data. However, load is typically not distributed evenly on all items stored by the system. Often, a majority of requests target only a relatively small number of items (e.g. by following a Pareto distribution). Since PRBR manages separate command sequences for different items, it is possible to use c-sets in only some of them. To reduce storage overhead, it is sensible to use them for only highly requested items for which conflicts of commutative writes are expected to be common.

The conditional use of c-sets, as well as the use of hard limits were not implemented in the scope of this thesis. In addition, fast writes were not considered as well. Use cases in which both fast writes and c-sets are useful for the same item seem limited. Fast writes are applicable if a proposer is confident that it can propose multiple commands sequentially without intermediate commands from another proposer interfering with it. In contrast, c-sets are useful when concurrent commutative writes are expected. Using both methods at the same time seems contradictory. However, it is possible to use them both on the same command sequence, just not simultaneously. The protocol flow remains largely unchanged for commands that are proposed without a command class. Each of these commands will be assigned a new round number, as it was the case in PRBR's unmodified state. Thus, a proposer that wants to submit a fast write command can do so by ensuring that this command is proposed without a class. Of course, the acceptors must detect that a fast write is in progress and change their behavior accordingly. Note that this is also just a theoretical consideration and was not implemented.

In the end, the potential of two proposers blocking each other when proposing commutative write commands for the same item still exists. However, such conflicts happen only if the c-set size limit is reached, which means that the total number of conflicts is expected to decrease. This should increase the throughput achievable during concurrent access of the same item. However, this speed-up is expected to be much lower than that of the modification made for commutative reads, which was discussed in Section 8.2, because the conflict potential was eliminated completely in this case.

8.5 Summary

Several modifications that optimize PRBR for concurrent, commutative commands were discussed in this chapter. Three scenarios were studied that each resulted in distinct changes to the protocol: commutative reads, reads that commute with write commands, and commutative write. This section gives a short summary of these changes.

Commutative Reads Examining the interference relationships of read commands revealed that reads do not interfere with any other command. It followed the insight that reads do not have to modify the read round of the respective acceptors, making them "invisible" to other commands.

By including the command type (read or write) in the messages of the round_request phase, acceptors are now able to respond to reads without modifying their state (i.e. rounds). Because of that, subsequent reads are processed in the same round, which eliminates all conflict potential between them. As a positive side effect, reads no longer block concurrently submitted write commands.

Reads Commuting with Write Two changes were discussed in this section. The first modification was based on the observation that proposers can ignore inconsistent reads rounds when processing read commands. Inconsistent read rounds indicate that a write command is in progress. However, due to the k-quorum property, it is impossible for a not seen write command to be chosen if the received write rounds are consistent.

Thus, reads can now be delivered when another proposer is negotiating rounds for a concurrent write command. It is also possible to deliver a read if some acceptors have already voted for a concurrent write command, as long as this set does not intersect the quorum of the read.

The second optimization of this section made use of pre-defined commutative relationships between a given read and write command. It turned out that a proposer can deliver a read as long as the newest write command that might have been chosen commutes with it. To decide if this is the case, it is only necessary for the proposer to examine the reply of the acceptor with the highest write round. This result was not intuitive and therefore required a formal proof (see page 53).

Commutative Writes The last scenario examined commutative writes. The key-insight here was that an arbitrary number of write commands can be proposed and learned in the same round as long as they all commute with each other. For that, the command sequences managed by PRBR were generalized to command set sequences. The assignment of command classes to commands allowed the modelling of commutative relationships between commands if they are transitive (and commutative).

As a result, concurrently proposed write commands with the same command class can now be proposed in the same round number. Since all such commands commute with each other, acceptors can vote for them in an arbitrary order.

However, this approach requires acceptors to temporarily store all commands that have been proposed in the current round number. To restrict the growth of the internal state of acceptors, it was necessary to introduce a (soft) size limit on command sets. Once this limit is reached, acceptors are forced to increment their rounds, which starts a new command set. As long as commutative write

commands are proposed within the size limit, no conflicts can occur. However, once the limit is reached and a new command set is started, conflicts can occur during this transition phase if a proposer receives an inconsistent state.

This modification of PRBR turned out to be the most complex because it required changes spanning all phases. Section 8.4.5 discussed some further optimizations of the command set approach on a theoretical level, including the incorporation of fast writes and the possibility of a hard size limit on command sets. However, these were not implemented in this thesis and are subject to future work.

Part III
Evaluation

9 Comparison with other Approaches

This chapter compares PRBR, including the modifications made in the context of this thesis, with other Paxos-based approaches that support commutative commands. The comparisons are made on a purely conceptual level. More meaningful results could have been achieved by implementing them as part of Scalaris' replication layer and comparing their performance in an experimental evaluation under different workloads. Due to their complexity, however, it was not feasible to implement them in the context of this thesis.

9.1 Generalized Paxos

Generalized Paxos [25], proposed by Leslie Lamport in 2004, closely resembles the original Paxos described in Chapter 4. However, it allows learning a sequence of commands with a single Paxos instance by use of command structures. Basically, a command structure can be seen as a directed graph whose nodes are the proposed commands and edges represent interference of two commands. This allows modeling arbitrary interference relationships between commands, making them more flexible than the command sets introduced in Section 8.4.2. Read and write commands are not distinguished by the protocol.

As long as two concurrently proposed commands do not interfere, they do not create conflicts and can be learned in four message delays during normal execution. Furthermore, if the quorum size is increased from a simple majority to two-thirds of all acceptors in the system, then it is possible to learn non-interfering commands in two message delays, which is optimal as argued by Lamport. If two commands create a conflict, at least four additional messages delays are needed to resolve it, similar to PRBR's WriteThrough mechanism.

However, Generalized Paxos has two major drawbacks in comparison to PRBR. First, ordering concurrent, interfering commands requires the existence of a stable leader process that can resolve the conflict. The disadvantages of such a leader were already discussed in Section 4.5. Most notably, the leader must receive every proposed command because it is not known beforehand when a conflict will occur. This makes the leader to a bottleneck of the system. Furthermore, it also represents a single point of failure. The system is not able to resolve any conflicts until the crash of a leader is detected. Only then is it possible to start the election of a new leader, which is, in turn, an instance of the consensus problem as well. However, the protocol can fall-back to standard Paxos in the mean time.

Generalized Paxos' second problem is that of increasing state and message sizes as new commands are proposed. New commands must be appended to the existing command structures, which are in turn included in messages. To prevent

unrestricted growth, the leader must submit a checkpoint command. Such a command interferes with all other commands and thus temporarily prevents all other commands from succeeding.

This is similar to the problem encountered by the use of command sets when modifying PRBR to support commutative write commands. However, due to their simpler structure, it is possible to start a new command set by incrementing round numbers without a dedicated checkpoint if the intermediate state is consistent. Otherwise, a WriteThrough will be started which is analogous to a checkpoint command.

In contrast to PRBR, which manages a register of command sequences, Generalized Paxos is designed for a single sequence containing all commands for all keys. In PRBR's case, only the commands proposed for the same key will be blocked by a WriteThrough. Thus, this lessens the impact of such a blocking command.

As of today, two notable extensions of Generalized Paxos exist. First, Fast Genuine Generalized Consensus [40] which reduced the number of message delays needed to recover from interfering commands by centering quorums around the coordinator (leader). Second, Multicoordinated Paxos [7] which makes use of multiple coordinators. Here, clients send their commands to a quorum of coordinators instead of one, which increases availability for non-interfering commands. However, it also relies on a stable leader to establish a consistent order when interfering commands arrive at coordinators in different order.

9.2 Egalitarian Paxos

Egalitarian Paxos (abbreviated as EPaxos) [33], proposed in 2013, is a Paxos variant that is designed to establish a single command sequence by making use of interference (or the absence thereof) between commands. It furthermore tries to solve the bottleneck of a single leader by assigning the replica that has received a command to its command leader[9]. This way, all replicas can act as the leader for a disjunct subset of submitted commands, distributing the load more evenly.

The protocols structure varies fundamentally from other Paxos variants, which is making direct comparisons difficult. Thus, the general outline of EPaxos will be sketched in this section.

As the main difference, EPaxos differentiates between *commit* and *execution* of a command. Is a command committed, then it is certain that it will be executed eventually. The execution phase handles the application of the command.

The commit protocol of EPaxos is divided into two phases: pre-accept and accept. In the pre-accept phase, the ordering constraints for a command *cmd* are established. Is *cmd* submitted to a replica (*cmd*'s command leader), it calculates the dependencies of *cmd* and sends this information to all other replicas. For that, each replica must maintain a command log of its seen commands. Each replica receiving the message updates its own command log and might amend the received dependency information before sending them back to the command leader. Receives the command leader responses from a fast quorum (which consists of roughly two-thirds of all replicas) with consistent information, then it can

[9]The concept of different roles does not exist in EPaxos. Thus, the generic term 'replica' is used instead of 'acceptor' or 'proposer' by its creators.

immediately commit the command by notifying the client and all replicas with a commit message. This is called a fast execution. Receives the command leader inconsistent information or less than a fast quorum of replica responses, the additional accept phase is needed to prevent inconsistencies in the dependencies across replicas.

For command execution, every replica creates a local dependency graph based on its command log and executes all its commands in the resulting order. Command execution is only performed once a client reads the replicated state.

Due to the separation of commit and execution of a command, clients do not know if a committed command was already executed. Furthermore, the execution order of two committed commands is unknown if they were not serialized by clients (i.e. one command is proposed after the other was committed by any replica). If this knowledge required, a read command must be atomically executed after the committed write command before delivering the result to the client.

If a replica wants to enforce the commit of a command, e.g. because it wants to execute it, or if it suspects that the command leader of a not yet committed command failed, a separate recovery procedure must be invoked. Basically, this procedure entails taking ownership of the command and subsequently continuing with the normal commit protocol without the possibility of a fast execution.

As with Generalized Paxos, Egalitarian Paxos manages only a single command sequence instead of independent sequences for each item. Committing a write command takes two message delays if the requirements of a fast execution are satisfied, otherwise four. Due to the separation of commit and execution, this does not change if two interfering commands are submitted concurrently. Is it required to know for certain if a command was executed, or if the result of the write command is of interest, then at least two additional messages delays are needed for the subsequent read. Note that the authors have not explicitly stated how a read is actually performed. It seems, however, that read commands have to be committed in the same way as write commands with the subsequent command execution phase.

By using a dependency graph, EPaxos allows to model arbitrary interference relations, in contrast to the c-set approach that required transitivity for commands targeting the same key. However, building the dependency graph requires each replica to maintain a private command log of all seen commands, which increases the size of each replicas state over time as more commands are proposed. Mechanisms to safely prune the command log are not explicitly stated.

10 Experimental Evaluation

10.1 Hardware Setup

The *cumulus* cluster at the Zuse Institute Berlin was used for all experiments. The description of cumulus' hardware was adopted from [14].

The cluster consists of two Dell M1000 chassis, both holding 16 server blades each. Each chassis holds a switch of type Dell Force10 MXL, which provides 10 Gbit/s point-to-point connections for all nodes within the chassis. The two chassis are connected by two 40 Gbit/s links. Every node houses two Intel Xeon E5-2630

v3 (20M Cache, 2.4 GHz) with eight cores each, for a total of 16 cores and 64 GB main memory.

Load generation was performed on a different host, connected to each chassis with 10 Gbit/s links using a Dell N4032F switch. This host also has two Intel Xeon E5-2630 v3 CPU's, but 128 GB of main memory.

10.2 Methodology

At the beginning of each measurement, a new Scalaris instance is started. Each instance consists of four Scalaris nodes, with each node deployed on a separate machine of the same chassis. A replication factor of four is chosen, which means that each Scalaris node contains one replica of each item. Load generation is performed by the benchmarking tool basho-bench [10].

Measurements are carried out over a duration of six minutes. However, the first and last 30 seconds of each measurement are discarded to prevent abnormal behavior during startup or shutdown of Scalaris or basho-bench from interfering with the results.

The achieved throughput is used as the primary performance metric in all experiments. basho-bench is configured to log histograms of the performed operations in one-second intervals, which means that each measurement results in 300 data points. For analysis, the mean with a 95% confidence interval assuming a normal distribution is used (no visible confidence interval indicates that it is so small that is obscured by the mark in the diagram).

The changes made to PRBR all concern concurrent command that access or modify the same item. Therefore, all operations during all experiments are performed on a single item. This constitutes the worst case for PRBR because concurrent commands targeting different do not create conflicts.

For simplicity, the item's value is a single integer that can be incremented or read by commands that are submitted by basho-bench work processes. This prevents any eventual overhead caused by, for example, computationally intensive filters, to interfere with the results.

Each basho-bench worker is a separate Erlang process that can be configured to submit requests at a fixed rate, or the maximum rate possible. For all measurements, the maximum option is used, which means that as soon as a worker receives a reply from a request, it submits the next one. The number of worker processes used will be stated for each measurement.

The raw data obtained from basho-bench, as well as a snapshot of Scalaris with all modified versions of PRBR are available at GitHub[11].

10.3 Measurements

10.3.1 Commutative Reads

This section examines how the number of concurrent clients (i.e. basho-bench workers) affects the number of read requests that can be handled per second. Naturally, the modification made in PRBR in regards to write commands

[10] `http://docs.basho.com/riak/kv/2.2.3/using/performance/benchmarking/`

[11] `https://github.com/rumoe/thesis`

should have no effect here, since no write commands are submitted. Thus, only the unmodified version of PRBR will be compared with the modification for commutative reads discussed in Section 8.2.

To summarize the achieved results, the modification completely eliminated the conflict potential of two concurrently proposed reads commands by not incrementing read round numbers when an acceptor receives a read command. Therefore, it is expected that the number of handled requests initially grows nearly linear. Of course, linear growth can not be sustained by using constant resources. Every proposed command needs some minimal amount of computational work and bandwidth. Therefore, the number of handled requests must eventually approach some limit.

In contrast to that, conflict potential exists for every pair of concurrently submitted commands in PRBR's unmodified version. The more read commands are proposed concurrently, the higher the likelihood of a conflict. Therefore, the number of requests is expected to grow only for a very small number of clients, albeit not nearly linear. After a certain threshold, repeated conflicts should increase in number, causing the number of handled requests to decrease.

The results of the experiment are shown in Figure 10.1. In general, the results are as expected. The handled requests in PRBR's modified version (blue) increase monotonically. A slight decrease in the graph's slope is noticeable, suggesting that some limit will be approached asymptotically for a higher number of clients. The performance of the unmodified version (red) decreases if more than two clients are used. However, it is surprising that the performance of both versions is equal for two clients. This might suggest that some side effect caused by this particular benchmark setup serializes the submitted requests, therefore preventing conflicts from occurring. To investigate further, this measurement was repeated

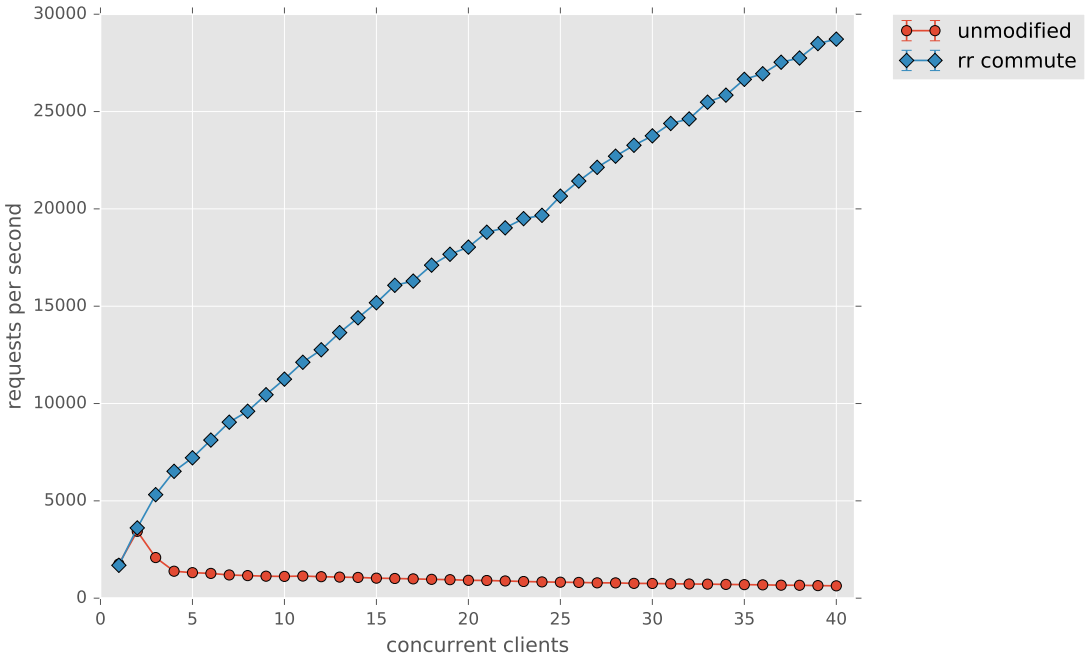


Figure 10.1: Commutative reads with an increasing number of concurrent clients.

with conflict-logging enabled. This revealed a small number of conflicts occurring (roughly 50 conflicts per second). It seems that messages sent from a proposer to all acceptors arrive within a small time frame in this configuration, which leaves only limited room for different messages to arrive in-between.

Note that the logging of conflicts might affect the number of conflicts that occur because it introduces some additional overhead. To prevent this from happening, conflict logging will not be performed otherwise

10.3.2 Commutative Writes

The next experiment focuses on concurrent, commutative writes on a single item. Similar to the previous measurement, the number of requests per second is plotted for an increasing number of concurrent workers. All submitted write commands commute with each other, which means that the c-set size would grow indefinitely without the establishment of a size limit. Thus, the size limits of 1, 5, 10, 20 and 50 will be used and compared against PRBR's unmodified version. To reiterate, the established limit is not a hard-limit (see Section 8.4.5). C-set sizes might grow larger than this, which makes a size limit of 1 interesting to compare against the unmodified version.

Each time the size limit is reached, conflict potential exists due to the possibility of a proposer receiving an inconsistent write state. Thus, no monotonic increase of handled requests can be expected. Furthermore, resolving a conflict is much more expensive due to the WriteThrough mechanism. Therefore, it is expected that all plotted graphs will resemble that of PRBR's unmodified behavior for commutative reads (an initial performance increase with subsequent degradation), but with a more aggressive decline in handled requests as the number of clients is increased. Increasing the c-set size limit reduces the conflict potential, while it simultaneously increases the size of sent messages. Since bandwidth is a limited resource, the cost of larger c-sets should eventually outweigh the benefits of fewer conflicts. Therefore, it is expected that the performance of PRBR increases initially for larger c-sets, but will degrade if a too large size is chosen.

The results of the measurements are depicted in Figure 10.2.

The results match the expectations in general. However, some interesting observations can be made. First, the performance of the unmodified version degrades as soon as a second client is introduced. This is in direct contrast to the read performance of the previous experiment. This can not be solely explained by the more expensive conflict resolution. By examining the protocol, it is obvious that conflicts are also much more likely to happen. A write will be blocked if a k-quorum of acceptors made promises during the time in between the start of the writes round_request and write phase. This time frame spans at least two message delays, which is much longer than the time frame in which reads could block each other.

The effects of using a soft-limit can be seen when comparing $L = 1$ to the unmodified version. Initially, the performance is slightly lower for a single client due to the slightly larger messages. However, the more concurrent clients are available, the higher $L = 1$'s relative performance increase compared to the unmodified version is. Since more commands are proposed concurrently, this likelihood is

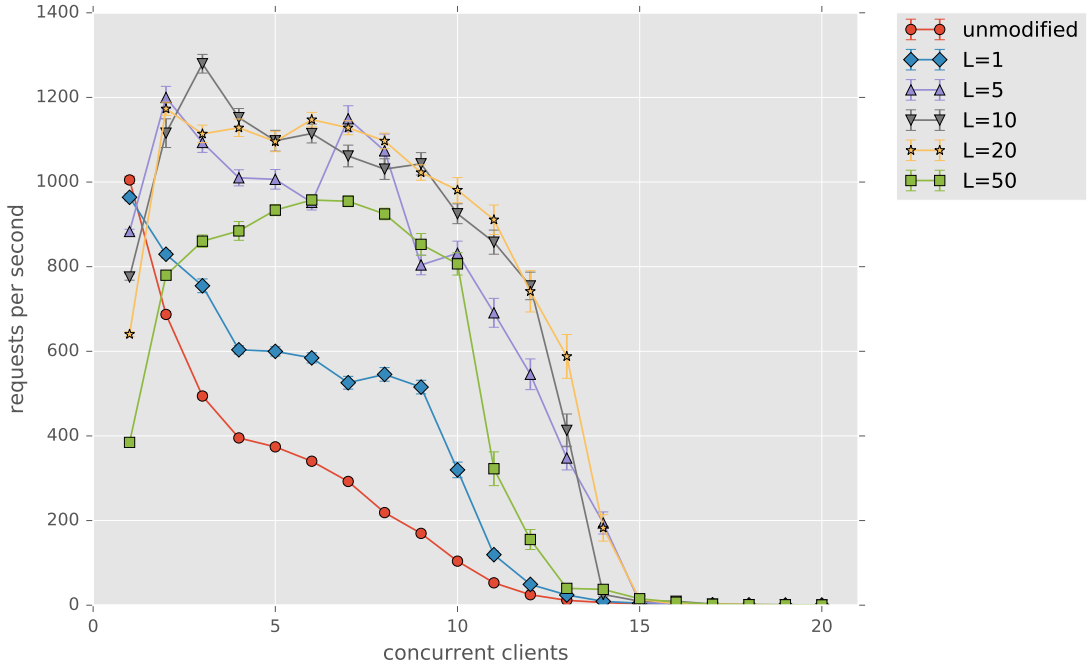


Figure 10.2: Commutative writes with different c-set size limits and an increasing number of concurrent clients.

higher that some pass the round_request phase in the same round. Thus, conflict potential is reduced and the number of requests grows.

This effect is even more pronounced for larger c-sets. For larger c-sets, the performance degradation when using a single client increases. All configurations (except $L = 1$) behave roughly the same: an initial performance increase for a small number of concurrent clients, followed by some range in which the performance remains fairly unchanged, and a sharp performance decrease around 9 to 10 concurrent clients.

Examining the raw data reveals that the achieved performance varies strongly for a higher number of clients. A possible reason for that is the use of WriteThroughs. WriteThroughs are write commands that do not commute with any other command. If a high number of clients is present, the probability of a client receiving an inconsistent write state due to a WriteThrough is also high. This client, in turn, will also start a WriteThrough. This can potentially trigger a cascade of WriteThroughs which will considerably slow down the system. Once this state is resolved, the PRBR returns to a much higher request rate. This indicates that the current write conflict resolution mechanism is not designed to handle a consistent stream of commands from multiple clients.

In addition, this strongly suggests that the obtained data has no normal distribution for a high number of clients. Thus, the plotted confidence intervals are not reliable here.

Out of the tested parameters, a c-set size limit of 20 seems to be performing best in most of the configurations. For the higher limit of 50, the increase in message sizes outweighs the benefit of fewer conflicts. Thus, it performs consistently worse.

$L = 5$ seems to be the best choice overall out of the tested configurations. Its performance is comparable to that of $L = 10$ and $L = 20$, albeit with higher

fluctuations. Due to the small size limit, however, the performance hit for non-concurrent access is relatively small. In addition the amount of data each acceptor must hold for $L = 5$ is considerably smaller than for the larger sizes. Storage size was not a performance metric in these benchmarks. However, it is sensible in real-life scenarios to keep storage overhead as small as possible.

10.3.3 Mix of Commutative and non-Commutative Writes

In the previous experiment, all submitted write commands commuted with each other. Of course, this is not always the case. This measurement focuses on PRBR's behavior for different percentages of commutative and non-commutative write commands using a fixed number of concurrent clients. Based on the result of the previous experiments, a c-set size limit of 5 will be used and compared with PRBR's unmodified state. The number of concurrent clients is set to four so that each basho-bench worker can send requests to a separate Scalaris node.

Since PRBR's unmodified version has no concept of commutativity, a constant throughput is expected for every configuration. For the version using c-sets, however, fewer conflicts should occur with an increasing percentage of commutative commands. By making some simplifying (and optimistic) assumptions, a prediction about the relationship between commutative command percentage and throughput can be made.

Assume that all clients submit their write commands at exactly the same time and in the same interval. Furthermore, assume that all write commands take the same time to process. No conflict occurs if all commands commute with each other. Let x be the percentage of commutative write commands. Then, the probability that no conflict occurs for a set of four commands is x^4. Otherwise, a conflict occurs that must be resolved first. In other words, the submitted commands can be processed faster with a probability of x^4. This suggests the existence of a quartic relationship between throughput and the percentage of commutative write commands. By examination of the previous results, a throughput of roughly 390 and 1010 requests per second is expected for 0% or 100% of commutative writes, respectively. This yields the function $f(x) = 390 + 620x^4$, which will be plotted as well.

The assumptions made are optimistic. In the used setup, clients do not synchronize their submissions and do not wait for each other. This means that during the time a write command is processed, more than three other commands can be submitted by the other clients. This reduces the probability of succeeding without conflict. Therefore, this function should be considered as an upper bound on the achieved performance.

Figure 10.3 depicts the results of the experiment. Unsurprisingly, the unmodified version of PRBR shows a constant performance with only minor fluctuations. The general shape of the modified version's graph matches that of the predicted upper bound. Due to the aforementioned argument, the achieved throughput is considerably lower for higher percentages of commutative write commands than was calculated by the naively derived function.

The main insight from this experiment is, that a significant portion of write commands must be commutative to have a noticeable positive effect on the throughput. A change of the percentage of commutative writes from 0% to 60%

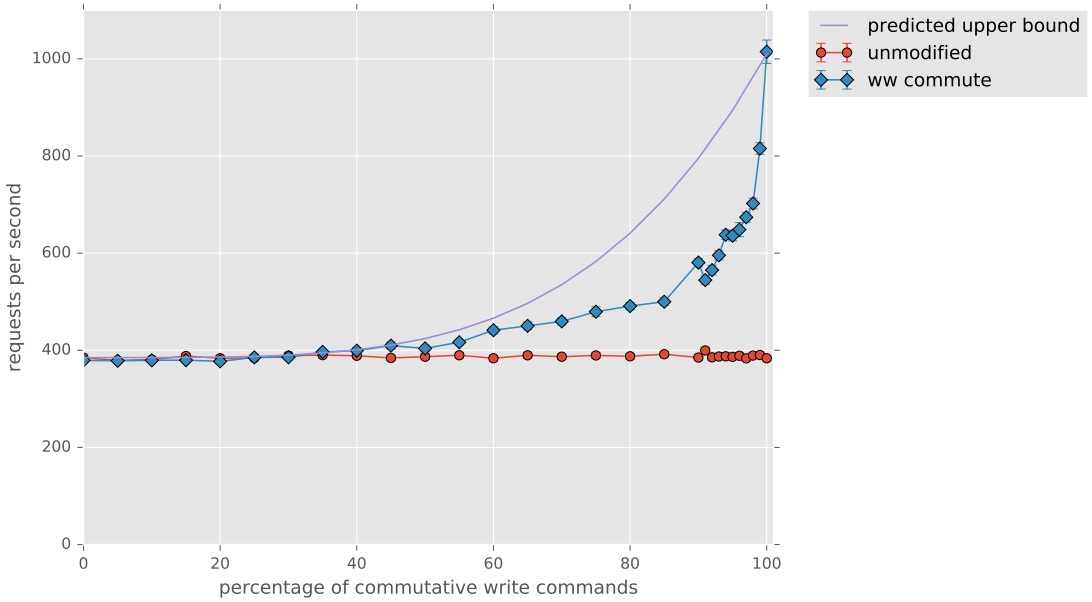


Figure 10.3: Mix of commutative and non-commutative writes using four concurrent clients.

achieves only a roughly 10% higher throughput, whereas the change from 80% to 100% doubled the observed throughput. This effect is expected to be even more pronounced for more concurrent clients since a single conflict has the potential to block more submitted commands from succeeding.

10.4 Mix of Read and Write Commands

In the last experiment, clients submit a mix of read and write commands. As with the previous experiment, four clients will be used and a c-set size limit of $L = 5$.

Six configurations that are making use of different commutative relationships will be compared.

unmodified PRBR's unmodified state to establish the baseline performance.

rr All read commands commute with each other.

passive This incorporates commutative reads and Change 2 of Section 8.3 (reads can ignore inconsistent read rounds caused by writes). This includes all modifications that are *always* applicable, i.e. without assuming specific relationships between commands, hence the name 'passive'.

passive+rw The same as passive, but all reads commute with all writes (writes, however, do not commute with each other).

passive+ww The same as passive, but all writes commute with each other.

passive+rw+ww All submitted commands commute with each other.

Write commands are more expensive than read commands. Thus, the performance of all configurations should increase as the percentage of read command

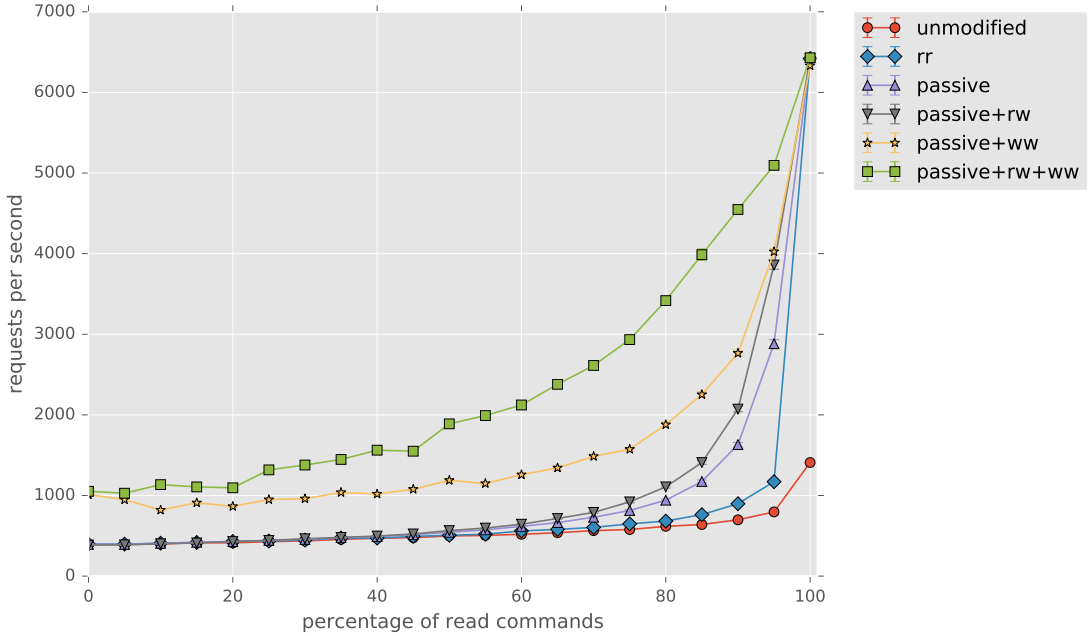


Figure 10.4: Mix of reads and writes using four concurrent clients.

increases. Furthermore, the order in which they are listed reflects the expected performance in relation to each other. The unmodified version of PRBR should be the slowest, whereas the configuration without non-commutative commands should perform best. The reason for that is the reduced conflict potential as more and more commands commute. Figure 10.4 depicts the result of the experiment.

All configurations demonstrate superlinear growth. The ordering in respect to their performance matches the expectations. Their performance for 0% and 100% matches with the results of the previous experiments. Since all but unmodified PRBR can handle concurrent reads without conflicts, their achieved throughput converges to the same value.

The way this experiment was set up allows it to group the used configurations into those with commutative write commands and those without them.

For all configurations without commutative writes, only a minor performance increase is noticeable if less than 50% of submitted commands are reads. This is consistent with the previous experiment: Even if there are two or more concurrently proposed read commands, a single concurrent write is enough to cause conflicts with both of them. The differences get more pronounced as the rate of read commands approaches 100%.

The *passive* graph is probably the most important since it includes all modifications that can be applied independently of the submitted commands. These modifications turn out to be effective for a high percentage of read commands, increasing the throughput by up to a factor of roughly four. The impact of only a few writes on the achievable performance is profound. Only five percent of writes commands are enough to more than half the achieved throughput compared to no writes at all.

It is unsurprising that configurations exploiting commutative writes perform better than those that do not. Of course, this performance gap shrinks as the

amount of write commands decreases. The only configuration which experienced a significant speedup by the 50% mark is the configuration in which all commands commute with each other. On the first glance, it might be surprising that this configuration experiences superlinear growth as well. If the number of write commands decreases linear, the number of messages decreases linear as well. However, less writes also mean that c-sets are filled slower. Therefore, fewer new c-sets must be started, which decreases the potential for conflicts to occur.

10.5 Summary of the Results

The modifications made to PRBR improved its performance throughout nearly all tested workloads. Only when handling serialized writes the measured performance degraded due to the additional data that had to be included in messages.

The achieved performance increase was negligible for workloads in which the percentage of commutative commands was not high. This effect is expected to be even more pronounced for a higher number of concurrently proposed commands. However, a noticeable improvement on the measured throughput could be observed for read-heavy workloads.

It was possible to completely remove the conflict potential in read-only scenarios, which completely changed the relationship between the number of concurrent clients and throughput. Instead of a decrease in throughput, near-linear growth was observed for the tested configuration.

Furthermore, it was shown that PRBR is not designed to handle a high number of concurrent write commands trying to modify the same item. For an increasing number of clients, conflicts became increasingly likely, which caused the protocol to eventually livelock. PRBR's extension to c-sets alleviated this problem slightly. Depending on the c-set size limit, up to 10 clients were able to propose write commands without major performance degradation compared to a single client. However, this came at the cost of a higher protocol complexity and reduced performance when using a single client.

Part IV
Conclusion and Future Work

11 Conclusion

This work provided the first in-depth description of the current state of PRBR, a Paxos-based algorithm that can establish in-place consensus of any number of independent command sequences. In addition, a number of informal proof sketches that reasoned about the correctness of PRBR were presented.

The main contribution of this thesis constituted the extension of PRBR to exploit commutativity of concurrently submitted commands as part of the same command sequence. Several strategies – depending on the types of the submitted commands – were outlined.

In a short discourse, two other approaches were discussed and compared on a theoretical level which revealed some pros and cons between them.

In the last part of this thesis, the modifications made to PRBR were compared to PRBR's unmodified state in an experimental evaluation by using throughput as the primary performance indicator. The experiments showed that, in general, these changes successfully improved the throughput in scenarios in which a significant portion of the submitted commands commuted with each other. The most significant performance improvement was achieved for a read-only scenario. Here, all conflict potential was eliminated, which resulted in a near-linear scaling for the tested number of concurrent clients.

12 Future Work

PRBR can still be improved upon. Furthermore, the modifications made to it to support commutative write commands only represent a proof-of-concept. Thus, there is still a lot of interesting work left to do.

Conflict Resolution PRBR's current conflict resolution mechanism is based on WriteThroughs, which synchronize the state of replicas by using a predefined write command. As shown in Section 7.5, WriteThroughs might cause some write commands to be applied twice if this is not prevented by a ContentCheck. Of course, this is often not desirable. Further work must be done to improve the conflict resolution mechanism so that this problem can be prevented.

C-Sets It was shown that commutativity of write commands can be exploited by extending command sequences to command set sequences. Due to the nature of c-sets, it is only possible to model transitive relationships that way. Other approaches like Egalitarian Paxos or Generalized Paxos do not require transitivity but are considerably more complex. In PRBR's case,

however, only concurrent write commands (since conflicts involving reads are handled differently) that operate on the same key must be considered, whereas the other approaches must consider all commands. It is of interest to investigate if use-cases exist that justify the added protocol complexity to support non-transitive commutativity relationships between write commands. If this is the case, it might be useful to incorporate some ideas of the other approaches to relax the transitive requirement.

The conflict potential could not be eliminated completely in the scenario where all submitted write commands commute with each other. The experimental evaluation showed that too many concurrent writes can still cause a livelock in the protocol. Further research is needed to explore ways to further reduce conflict potential in this case.

Furthermore, some possible extensions of the c-set approach were mentioned in Section 8.4.5. These include the establishment of hard size limit, the incorporation of the fast write mechanism, and the restricted use of c-sets for only some command sequences managed by PRBR. Further work must be done to evaluate the practical feasibility and effectiveness of these ideas.

Hardware Focus All considerations up until this point focused on PRBR solely on an algorithmic level. Due to the fundamental and widely applicable nature of the consensus problem, however, it is of interest to further improve efficiency by making use of the capabilities of innovative hardware. Potential optimizations include the exploitation of multicast and reduce operations of modern interconnects such as InfiniBand and OmniPath to reduce the number of messages and latency, or the use of techniques such as RDMA (remote direct memory access) in conjunction with NVRAM (non-volatile RAM) for more efficient access to the distributed state. An upcoming DFG (Deutsche Forschungsgesellschaft) project[12] starting end of 2017 as part of the SPP2037[13] plans to address some of these issues.

[12] `https://www.dfg-spp2037.de/re1389-10/`

[13] `http://www.dfg.de/foerderung/info_wissenschaft/2016/info_wissenschaft_16_26/index.html`

Bibliography

[1] Marcos Aguilera, Wei Chen, and Sam Toueg. "Failure detection and consensus in the crash-recovery model". In: *Distributed Computing* (1998), pp. 231–245.

[2] Michael Barborak, Anton Dahbura, and Miroslaw Malek. "The Consensus Problem in Fault-tolerant Computing". In: *ACM Comput. Surv.* 25.2 (June 1993), pp. 171–220. ISSN: 0360-0300.

[3] Zuse Institute Berlin. *Scalaris - Distributed Scalable Key-Value Store with Transactions*. Accessed: 08.06.2018. URL: `http://www.zib.de/projects/scalaris-distributed-scalable-key-value-store-transactions`.

[4] Zuse Institute Berlin. *Scalaris - Users and Developers Guide*. 2016. URL: `https://github.com/scalaris-team/scalaris/blob/master/user-dev-guide/main.pdf`.

[5] Mike Burrows. "The Chubby lock service for loosely-coupled distributed systems". In: *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association. 2006, pp. 335–350.

[6] Christian Cachin, Rachid Guerraoui, and Luis Rodrigues. *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[7] Lásaro Jonas Camargos, Rodrigo Malta Schmidt, and Fernando Pedone. "Multicoordinated Paxos". In: *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. ACM. 2007, pp. 316–317.

[8] Tushar D Chandra, Robert Griesemer, and Joshua Redstone. "Paxos made live: an engineering perspective". In: *Proceedings of the twenty-sixth annual ACM symposium on Principles of distributed computing*. ACM. 2007, pp. 398–407.

[9] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. "The weakest failure detector for solving consensus". In: *Journal of the ACM (JACM)* 43.4 (1996), pp. 685–722.

[10] James C. Corbett et al. "Spanner: Google's globally distributed database". In: *ACM Transactions on Computer Systems (TOCS)* 31.3 (2013), p. 8.

[11] Giuseppe DeCandia et al. "Dynamo: Amazon's Highly Available Key-value Store". In: *SIGOPS Oper. Syst. Rev.* 41.6 (Oct. 2007), pp. 205–220. ISSN: 0163-5980.

[12] Carole Delporte-Gallet et al. "From crash-stop to permanent omission: Automatic transformation and weakest failure detectors". In: *International Symposium on Distributed Computing*. Springer. 2007, pp. 165–178.

[13] Hao Du and David J. St. Hilaire. *Multi-Paxos: An Implementation and Evaluation*.

[14] Jens Fischer. "Evaluating the Scalability of Scalaris". MA thesis. Freie Universität Berlin, 2016.

[15] Michael J Fischer. "The consensus problem in unreliable distributed systems (a brief survey)". In: *International Conference on Fundamentals of Computation Theory*. Springer. 1983, pp. 127–140.

[16] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson. "Impossibility of Distributed Consensus with One Faulty Process". In: *J. ACM* 32.2 (Apr. 1985), pp. 374–382. ISSN: 0004-5411.

[17] Ali Ghodsi, Luc Onana Alima, and Seif Haridi. "Symmetric replication for structured peer-to-peer systems". In: *Databases, Information Systems, and Peer-to-Peer Computing*. Springer, 2007, pp. 74–85.

[18] 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), pp. 51–59.

[19] Rachid Guerraoui and Luis Rodrigues. *Introduction to reliable distributed programming*. Springer Science & Business Media, 2006.

[20] Maurice P Herlihy and Jeannette M Wing. "Linearizability: A correctness condition for concurrent objects". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 12.3 (1990), pp. 463–492.

[21] Patrick Hunt et al. "ZooKeeper: Wait-free Coordination for Internet-scale Systems." In: *USENIX annual technical conference*. Vol. 8. 2010, p. 9.

[22] Bjorn Kolbeck et al. "Flease-lease coordination without a lock server". In: *Parallel & Distributed Processing Symposium (IPDPS), 2011 IEEE International*. IEEE. 2011, pp. 978–988.

[23] Leslie Lamport. "Byzantizing Paxos by refinement". In: *International Symposium on Distributed Computing*. Springer. 2011, pp. 211–224.

[24] Leslie Lamport. "Fast Paxos". In: *Distributed Computing* 19.2 (2006), pp. 79–103.

[25] Leslie Lamport. *Generalized consensus and Paxos*. Tech. rep. Technical Report MSR-TR-2005-33, Microsoft Research, 2005.

[26] Leslie Lamport. "Paxos made simple". In: *ACM Sigact News* 32.4 (2001), pp. 18–25.

[27] Leslie Lamport. "Specifying concurrent program modules". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 5.2 (1983), pp. 190–222.

[28] Leslie Lamport. *The Part-Time Parliament*. Accessed: 26.05.2017. URL: `http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#lamport-paxos`.

[29] Leslie Lamport. "The Part-time Parliament". In: *ACM Trans. Comput. Syst.* 16.2 (May 1998), pp. 133–169. ISSN: 0734-2071.

[30] Leslie Lamport. "Using time instead of timeout for fault-tolerant distributed systems." In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 6.2 (1984), pp. 254–280.

[31] Leslie Lamport and Mike Massa. "Cheap Paxos". In: *Dependable Systems and Networks, 2004 International Conference on*. IEEE. 2004, pp. 307–314.

[32] Nancy A. Lynch. *Distributed algorithms*. Morgan Kaufmann, 1996.

[33] Iulian Moraru, David G Andersen, and Michael Kaminsky. "There is more consensus in egalitarian parliaments". In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. ACM. 2013, pp. 358–372.

[34] Hiroya Nagao and Kazuyuki Shudo. "Flexible routing tables: Designing routing algorithms for overlays based on a total order on a routing table set". In: *Peer-to-Peer Computing (P2P), 2011 IEEE International Conference on*. IEEE. 2011, pp. 72–81.

[35] Diego Ongaro and John K Ousterhout. "In Search of an Understandable Consensus Algorithm." In: *USENIX Annual Technical Conference*. 2014, pp. 305–319.

[36] Fred B. Schneider. "Implementing fault-tolerant services using the state machine approach: A tutorial". In: *ACM Computing Surveys (CSUR)* 22.4 (1990), pp. 299–319.

[37] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. *Chord#: Structured Overlay Network for Non-Uniform Load-Distribution*. 2005.

[38] Thorsten Schütt, Florian Schintke, and Alexander Reinefeld. "Scalaris: Reliable Transactional P2P Key/Value Store". In: *Proceedings of the 7th ACM SIGPLAN Workshop on ERLANG*. ERLANG '08. Victoria, BC, Canada: ACM, 2008, pp. 41–48. ISBN: 978-1-60558-065-4.

[39] Ion Stoica et al. "Chord: A scalable peer-to-peer lookup service for Internet applications". In: *ACM SIGCOMM Computer Communication Review* 31.4 (2001), pp. 149–160.

[40] Pierre Sutra and Marc Shapiro. "Fast Genuine Generalized Consensus". In: *Reliable Distributed Systems (SRDS), 2011 30th IEEE Symposium on*. IEEE. 2011, pp. 255–264.

[41] Paolo Viotti and Marko Vukolić. "Consistency in non-transactional distributed storage systems". In: *ACM Computing Surveys (CSUR)* 49.1 (2016), p. 19.