Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 136

Systems Group, Department of Computer Science, ETH Zurich

Consensus on a multicore machine

by

Roni Häcki

Supervised by

Prof. Timothy Roscoe
Stefan Kaestle
Moritz Hoffmann

March 2015–September 2015

inf | Informatik
Computer Science

Abstract

The communication latency within a multicore machine is almost non existent and non uniform. The reduced propagation time changes the requirements for a consensus protocol from reducing the number of communication rounds to reducing the number of messages. Furthermore, a multicore machine features clusters of cores within which consensus is easier and faster to solve. By running different protocols within clusters and between clusters we can solve consensus with a hierarchy of agreements.

In this thesis, we investigate the problems that consensus on a multicore machine faces and propose solutions. In a second part, we present an agreement framework that hierarchically composes different consensus protocols and shared memory according to the performance characteristics of the underlying multicore machine.

Contents

1 Introduction and Motivation **8**

2 Background **10**
2.1 Barrelfish . . . 10
2.1.1 CPU Driver . . . 10
2.1.2 Dispatcher . . . 11
2.1.3 Capabilities . . . 11
2.1.4 Inter-Dispatcher Communication . . . 12
2.1.5 Waitsets . . . 12
2.2 Multicore Characteristics . . . 13
2.3 Failure Model and Failure Domain . . . 13
2.4 Shared Memory vs. Message Passing . . . 14

3 Existing Consensus Protocols **15**
3.1 Two Phase Commit (2PC) . . . 15
3.2 Paxos and Multi-Paxos . . . 17
3.3 1Paxos . . . 20
3.4 Raft . . . 22
3.5 Evaluation . . . 27

4 Improvement for Multicore **28**
4.1 Consensus Inside . . . 28
4.2 Micro Benchmarks . . . 29
4.2.1 Setup . . . 29
4.2.2 Response Time . . . 29
4.2.3 Message Processing . . . 30
4.2.4 Wait Time . . . 32
4.2.5 Ping Pong Benchmark . . . 33
4.3 Tree Broadcast . . . 34
4.4 Protocol Implementation Overhead . . . 35
4.5 Selection Overhead . . . 35

5 Framework Design **36**
5.1 Composability . . . 36
5.1.1 Performance . . . 36
5.1.2 Failure . . . 37
5.2 Interfaces . . . 37
5.2.1 Replica Interface . . . 37
5.2.2 Internal Interface . . . 39
5.2.3 Client Interface . . . 40
5.2.4 Framework Interface . . . 41
5.3 Policy . . . 42
5.3.1 Protocol Choice . . . 42
5.3.2 Shared Memory . . . 42
5.3.3 Summary . . . 43

6 Implementation **44**
6.1 2PC . . . 44
6.1.1 Tree Broadcast . . . 44
6.1.2 Removing the Leader . . . 46
6.2 Raft . . . 46
6.3 1Paxos . . . 48
6.3.1 Acceptor Failure . . . 48
6.3.2 Leader Failure . . . 49
6.3.3 Flooding Message Channels . . . 50
6.4 Single-writer Multiple-reader Queue . . . 50
6.5 Broadcast Replication . . . 52
6.6 Composability . . . 52
6.7 Key Value Store . . . 53

7 Evaluation **55**
7.1 Setup . . . 55
7.2 Improvements for Multicore . . . 56
7.2.1 Tree Broadcast and Message Processing . . . 56
7.2.2 Barrelfish Implementations . . . 58
7.3 Failures . . . 59
7.4 Composition . . . 61
7.4.1 2PC . . . 61
7.4.2 2PC without Leader . . . 62
7.4.3 1Paxos . . . 63
7.4.4 Broadcast Replication . . . 65
7.5 Key Value Store . . . 66

8 Related Work **69**
8.1 Rex . . . 69
8.2 Composing Shared-Memory Algorithms . . . 69
8.3 Optimizing Collective Communication on Multicores . . . 70

9 Conclusion **71**
9.1 Future Work . . . 72
9.1.1 Failure Model . . . 72
9.1.2 Protocols . . . 72
9.1.3 Improvements for Multicore . . . 72
9.1.4 Network Layer . . . 72
9.1.5 Usage within Barrelfish . . . 72
9.1.6 Framework Cleanup . . . 73

A Two Phase Commit **74**
A.1 Consensus Inside Message Passing . . . 74
A.2 Barrelfish Message Passing . . . 75
A.3 Removing the Leader . . . 77
A.3.1 Benchmarks . . . 77

B Raft **79**
B.1 Problem Benchmarks . . . 80
B.2 Response Time . . . 82

C 1Paxos **84**

Bibliography **86**

List of Figures

2.1 The multikernel design model. Adapted from [4] 11

3.1 2PC: base case without conflict 16
3.2 Paxos: base case without failure 18
3.3 Multi-Paxos: base case without failure 19
3.4 1Paxos: base case without failure 21
3.5 Raft: base case without failure 26

4.1 Response time: comparing 1Paxos and 2PC using 8 replicas . . . 30
4.2 Micro benchmark: 2PC message types using 8 replicas 31
4.3 Micro benchmark: 1Paxos message types using 8 replicas 31
4.4 Micro benchmark: wait time using 8 replicas 32
4.5 Ping pong: send time between core 0 and cores 1-47 33
4.6 Tree broadcast: example of a one-sided tree 34

5.1 Shared memory: performance of writes w.r.t. number of cores . . 42

6.1 Tree broadcast: example of a one-sided radix 2 k-nomial tree . . 45
6.2 Raft: problem of additional messages 47
6.3 1Paxos: acceptor failure. Adapted from [9] 48
6.4 1Paxos: leader failure. Adapted from [9] 49
6.5 Shared memory queue: layout in memory 51

7.1 Multicore improvements: cycles per message type 2PC 56
7.2 Multicore improvements: cycles per message type 2PC using broadcast tree 57
7.3 Multicore improvements: throughput of implementations based on QC-libtask 57
7.4 Barrelfish: response time of different protocols w.r.t. number of clients 58
7.5 Barrelfish: throughput of different protocols w.r.t. number of clients 59
7.6 Failures: response time during recovery from a failure 60
7.7 Failures: performance w.r.t. number of failures recoverable 60
7.8 Composition: throughput of compositions of 2PC w.r.t. number of clients 61

7.9 Composition: response time of compositions of 2PC w.r.t. number of clients 62
7.10 Composition: throughput of compositions of 2PC without leader w.r.t. number of clients 62
7.11 Composition: response time of compositions of 2PC without leader w.r.t. number of clients 63
7.12 Composition: throughput of compositions of 1Paxos w.r.t. number of clients 64
7.13 Composition: response time of compositions of 1Paxos w.r.t. number of clients 64
7.14 Composition: throughput of compositions of broadcast replication w.r.t. number of clients 65
7.15 Composition: response time of compositions of broadcast replication w.r.t. number of clients 65
7.16 Key value store (CAS): write time w.r.t. number of clients 66
7.17 Key value store (spinlock): write time w.r.t. number of clients . . 67
7.18 Key value store (spinlock): throughput w.r.t. number of clients . 68

A.1 2PC: throughput w.r.t. conflict probability 78
A.2 2PC: response time w.r.t. conflict probability 78

B.1 Raft: throughput and additional number of messages w.r.t. the number of clients 81
B.2 Raft: throughput using 1 and 8 clients with a varying sleep time 81
B.3 Raft: response time w.r.t. number of clients 82
B.4 Raft: response time with sleep time between requests w.r.t. number of clients 83

List of Listings

5.1 Composability: replica interface 37
5.2 Composability: lower layer interface 39
5.3 Composability: client interface 40
5.4 Composability: replica to client communication interface 40
5.5 Composability: framework interface 41
6.1 Shared memory: single-writer multiple-reader queue interface . . 51
6.2 Key value store: simple implementation 53
6.3 Key value store: interface for clients 54
A.1 2PC: message struct definition 74
A.2 2PC: Flounder interface definition 75
A.3 2PC: abort message type 77
B.1 Raft: Flounder interface 79
C.1 1Paxos: Flounder interface 84

List of Tables

4.1 Micro benchmarks: cores assigned to NUMA nodes 29

7.1 Evaluation: benchmark machines 55
7.2 Evaluation: cores assigned to NUMA nodes 55

Chapter 1

Introduction and Motivation

In recent years, the core count of multicore systems increased and there is no end to this trend. One of the scaling problems for an operating system on a multicore machine is sharing data. The most widespread operating systems feature a single kernel where data is shared between all the cores. There is a general consensus that for scaling an operating system to a higher core count, data should no longer be shared but replicated.

Out of this reason, research operating systems as Barrelfish [24] started to see a multicore machine as a distributed system. Problems that are easy to solve when data is shared, become harder when distributed. Data that was previously shared, has to be held consistent between communication nodes. In distributed systems, consistent replication is often solved by consensus protocols.

Can we simply take any existing consensus protocol and run it on a multicore machine? A multicore machine has large differences to the traditional network setup. Communication within a multicore machine offers more guarantees and lower propagation time than communication over a network. If a consensus protocol does not consider the advantages of a multicore machine, the performance benefits are lost.

Consensus protocols are not only useful for replicating data, but also for tolerating failures. The current hardware model is not absolute. In the future, full verification of hardware may become too complex and failures have to be taken into account as part of the hardware model. If the hardware no longer guarantees failure safety, the software has to compensate. The operating system has to prevent the system from crashing because of a partial hardware failure.

Reaching consensus over a large number of replicas is not a cheap task. Replication over all cores of a multicore machine has to take into account the topology of the machine to be efficient. Certain clusters of cores share the same memory or even an L3 cache. If we define such a cluster as a failure domain, we

can replicate within a cluster by using shared memory and reduce the cost of replication. To prevent failures from crashing the whole machine, a consensus protocol that can tolerate failures is run between failure domains.

The goal of this thesis is to establish a framework for handling consensus on the Barrelfish operating system. The framework should provide a toolbox of different protocols and algorithms. Based on the topology of the multicore machine, the framework should compose shared memory with consensus protocols to yield optimal performance while still assuring fault tolerance for a given failure domain.

Chapter 2

Background

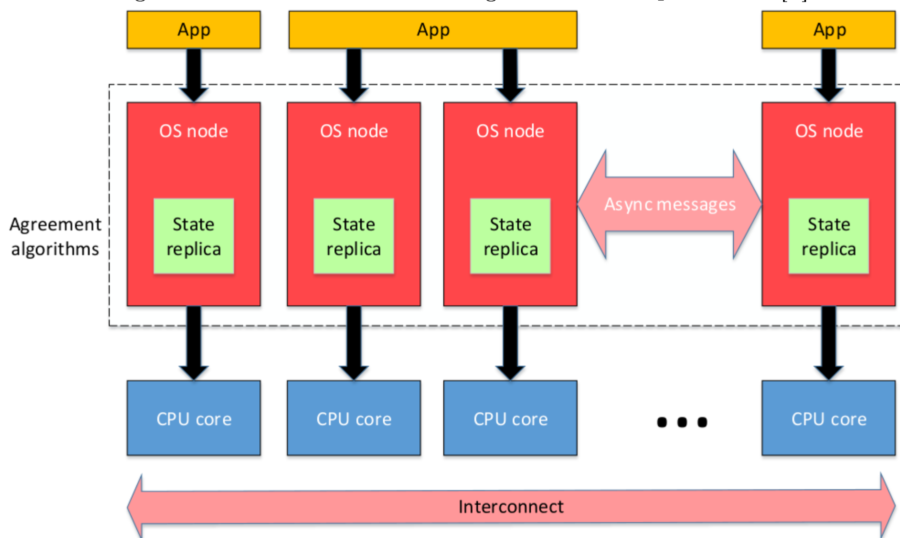
2.1 Barrelfish

Barrelfish [24] is a research operating system built at ETH Zurich. Barrelfish tries to tackle two upcoming problems. The first problem is scalability for multi- and many-core systems. The second is heterogeneity of system hardware. A large part of the design of Barrelfish is driven by these two problems.

2.1.1 CPU Driver

A new approach for designing an operating system is the multikernel [4]. Some time ago, operating system designers realised that for scaling to a large number of cores, sharing state between cores is becoming a bottleneck. In a multikernel each core runs a kernel or in Barrelfish terminology a CPU driver. Barrelfish tries to avoid shared memory and send explicit messages instead. The system itself is seen as a distributed system and not as a single large unit. The CPU driver is single threaded and non-preemptive. It abstracts very few resources similar to an exokernel [11]. The CPU driver provides hardware resource multiplexing, message passing, scheduling of dispatchers on the local core, as well as access of kernel objects and physical memory by using capabilities [19]. Other operating system services are either implemented in libraries or similar to microkernels [20] in servers that run as a user-space process. The main design blocks are shown in figure 2.1.

Figure 2.1: The multikernel design model. Adapted from [4]



2.1.2 Dispatcher

Dispatchers are the unit of kernel scheduling. In Barrelfish, dispatchers are implemented similar to scheduler activations [2]. The kernel manages a Dispatcher Control Block (DPC) for each dispatcher. The dispatcher itself schedules user-level threads. In contrast to a traditional kernel, the Barrelfish CPU driver uses upcalls to inform a dispatcher that it is scheduled. The remote endpoints used for messages passing are derived from the dispatcher's capability.

2.1.3 Capabilities

Capabilities [19] are a form of access control. The principle is inverse to Access Control List (ACL) where the permissions are attached to an object. Capabilities are keys enabling the subject to access a certain object. From user-space the capabilities referencing memory and other resources can not be directly accessed. Capabilities can only be manipulated by system calls. Therefore, the kernel prevents unauthorized accesses and security is enforced by the kernel. In Barrelfish, capabilities are typed ("untyped" being a type itself). All memory that is not claimed for bootstrapping is initially untyped. From the untyped capability, every other type of capability is formed by retyping (satisfying certain rules) and splitting the large untyped capability. The kernel does no dynamic memory allocation and can not run out of memory. The capability types and the rules of retyping are defined in a Domain Specific Language (DSL) called Hamlet [8]. Through Hamlet the capability system can use types freely.

2.1.4 Inter-Dispatcher Communication

The multikernel design heavily relies on message passing. A considerable amount of the performance of such an operating system depends on message passing. In Barrelfish there are two backends hidden behind a general message passing interface. Depending on if a message is core local or between cores, one of the two backends Local Message Passing (LMP) or User-level Message Passing (UMP) is used.

LMP

If two dispatchers on the same core are communicating, the LMP backend is used. Conceptual LMP is adapted from Lightweight Remote Procedure Call (LRPC) [5]. LMP requires endpoints to communicate and a channel has two endpoints (remote and local). The endpoints are retyped from the dispatcher of a process. Each endpoint contains buffers for message passing. The sender invokes the remote endpoint with the message to send. As a result of the invocation, the kernel copies the message into the associate buffer and makes the dispatcher runnable.

UMP

Inter core communication in Barreflish is implemented as a variation of User-Level Remote Procedure Call (URPC) [6] called UMP. Similar to LMP, a UMP channel requires an endpoint capability to communicate with. Messages are sent by shared memory in combination with polling and inter-process interrupts. The shared memory is split into sending and receiving buffer. Entries of a buffer are the size of a cache line. The sender writes into the buffer while the receiver is polling for new data. The buffers are circular and overwriting not yet processed data is prevented by an epoch flag bit.

Flounder

Flounder [3] is an interface definition language that generates code for message passing. Through Flounder the user can define message signatures based on common C types. Flounder generates all required functions from exporting functionality, over the binding process to initialize a channel up to marshalling the arguments and sending a message. It generates C code for both LMP and UMP and decides automatically which backend to use.

2.1.5 Waitsets

Waitsets are the main mechanism for event handling in Barrelfish. A message channel has to be registered to a certain waitset. Events on a message channel as receiving a message can be raised by calling `event_dispatch()` or `event_dispatch_non_blocking()`. A waitset maintains a queue of threads that

are waiting for events to be raised. An event is defined as a function pointer and a pointer to the arguments (closure).

2.2 Multicore Characteristics

What are the differences between running a consensus protocol over the network and running the protocol between cores of a machine? In general, communication channels between cores give more guarantees than the network between machines.

The largest difference is the propagation time. Over a network the propagation time takes up the largest chunk of time. On a multicore machine it is the contrary, the transmission time between cores is almost non existent. The dramatic change in transmission time can become a problem for adapting existing protocols to a multicore machine. Consensus protocols that are designed to work over a network, try to reduce the number of communication rounds. The main objective on a multicore machine, is to reduce the number of messages [9] since the dominating factor is the time to send and receive a message.

The communication channels on a multicore machine give the guarantee that no message is lost or dropped and a message can not be corrupted (as long as there are no hardware failures). Therefore, there is no need for checksums and resending messages. Building on these facts, we can save certain acknowledgement messages. Further, messages sent between cores have an upper bound on how long it takes until they arrive. A message sent over the network does not have an upper bound and can take an arbitrary time to arrive at the destination.

2.3 Failure Model and Failure Domain

The failure model is essential for a consensus protocol. If there are no failures in the system, simple protocols as Two Phase Commit (2PC) often yield good enough results. If crash failures are part of the model, protocols as Raft [23] and variants of Paxos [15] are frequently used. The most complicated case are Byzantine failures. Byzantine failures not only consider crashes of replicas but also malicious behaviour as injecting wrong information into the system. The more general the failure model, the more computationally intensive the consensus protocol.

In addition to the failure model, we have to define a failure domain. Currently, it is normal that if a single core of a CPU fails, the machine is shut down. If hardware failures become part of the computing model, shutting down the whole machine is not a reasonable choice. There are three granularities of failure domains we consider: machines, Non-Uniform Memory Access (NUMA) nodes, and cores. The smaller the failure domain granularity, the more replicas

are needed.

We did not want to start with the most complicated failure model at the beginning. We chose the simple model of crash-stop. In the crash-stop failure model, the unit that fails stops and does not execute any further operations. Further, there is no malicious behaviour. The failure domain we consider is a NUMA node. A NUMA node as the failure domain opens up the possibility to run a different protocol or algorithm within a node.

2.4 Shared Memory vs. Message Passing

In the past, shared memory and message passing were seen as a dual [18]. In recent years, message passing seems to be the preferable choice [4]. For each write to a shared memory location, the cache coherency protocol invalidates the cache entries of all the cores that share this particular cache line. As a result, the more nodes (and cores) that share a cache line, the slower the updates to shared memory get. Message passing as it is used in Barrelfish [24] allows the user only to allocate channels between a pair of cores. In this manner, the bus is not flooded from the cache coherency protocol and communication is made explicit. The only performance reduction for message passing when scaled to a larger number of cores is a queueing effect, since the CPU drivers are single threaded.

Chapter 3

Existing Consensus Protocols

The consensus problem requires multiple processes to agree on a single value. The more elaborate the failure model, the more communication is required to prevent an invalid agreement. A valid agreement fulfils the following requirements:

- **Termination:** every process decides on a value.
- **Validity:** if all processes propose the same value, all correct processes choose said value.
- **Integrity:** at most one value is chosen by a process and the value must have been proposed by some process.
- **Agreement:** all processes agree on the same value.

Consensus is a fundamental problem of distributed systems. Consensus is needed to prevent inconsistencies between replicas. Each consensus protocol has distinct characteristics. To compose different protocols, it is essential to understand the building blocks of the composition. In the following section, the most relevant protocols for this thesis are explained.

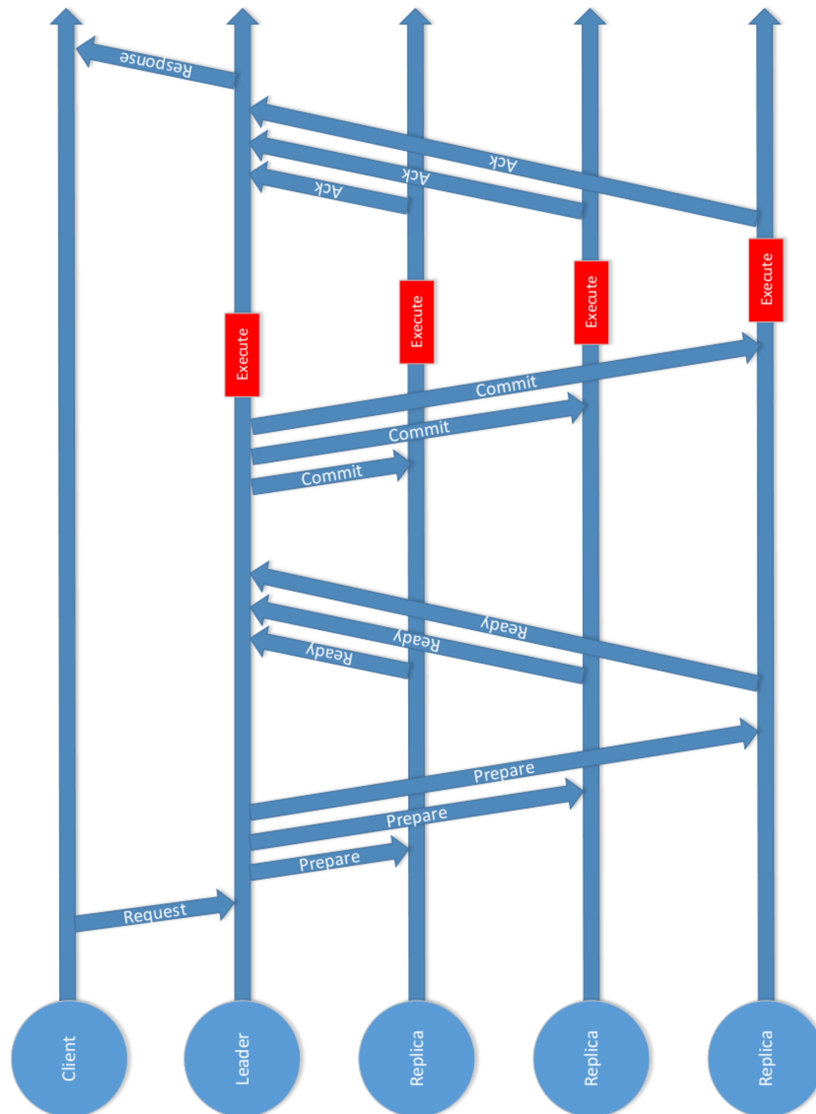
3.1 Two Phase Commit (2PC)

Two Phase Commit (2PC) is a well known consensus protocol that is based on a leader. It is often referred to in the context of database transactions. There are many variations of 2PC that are optimised for specific use cases. As the name suggests there are two phases:

- **Prepare Phase**: contacting all participating replicas and prepare them for the commit (and find conflicts if there are any).
- **Commit Phase**: after all replicas responded and no conflict is detected, the leader tells the replicas to commit. After a replica commits the changes, it sends an acknowledgement to the leader.

2PC can not handle failures because each replica has to respond to the leader. If either a replica or the leader fails, the agreement stalls. During normal operation the replicas communicate according to the pattern shown in figure 3.1.

Figure 3.1: 2PC: base case without conflict



If a conflict of two requests is detected by one of the replicas, the agreement has to be aborted. Instead of a *ready* message the replicas send an *abort* message.

If the leader receives any *abort* message, it broadcasts an *abort* message to all the replicas to avoid stalling further agreements.

Another version of 2PC removes the leader and lets the clients directly broadcast the *prepare* messages to the replicas. By removing the leader, the maximal throughput of the system should increase. A disadvantage is an increased conflict probability since multiple clients suggest values and not only a single leader.

3.2 Paxos and Multi-Paxos

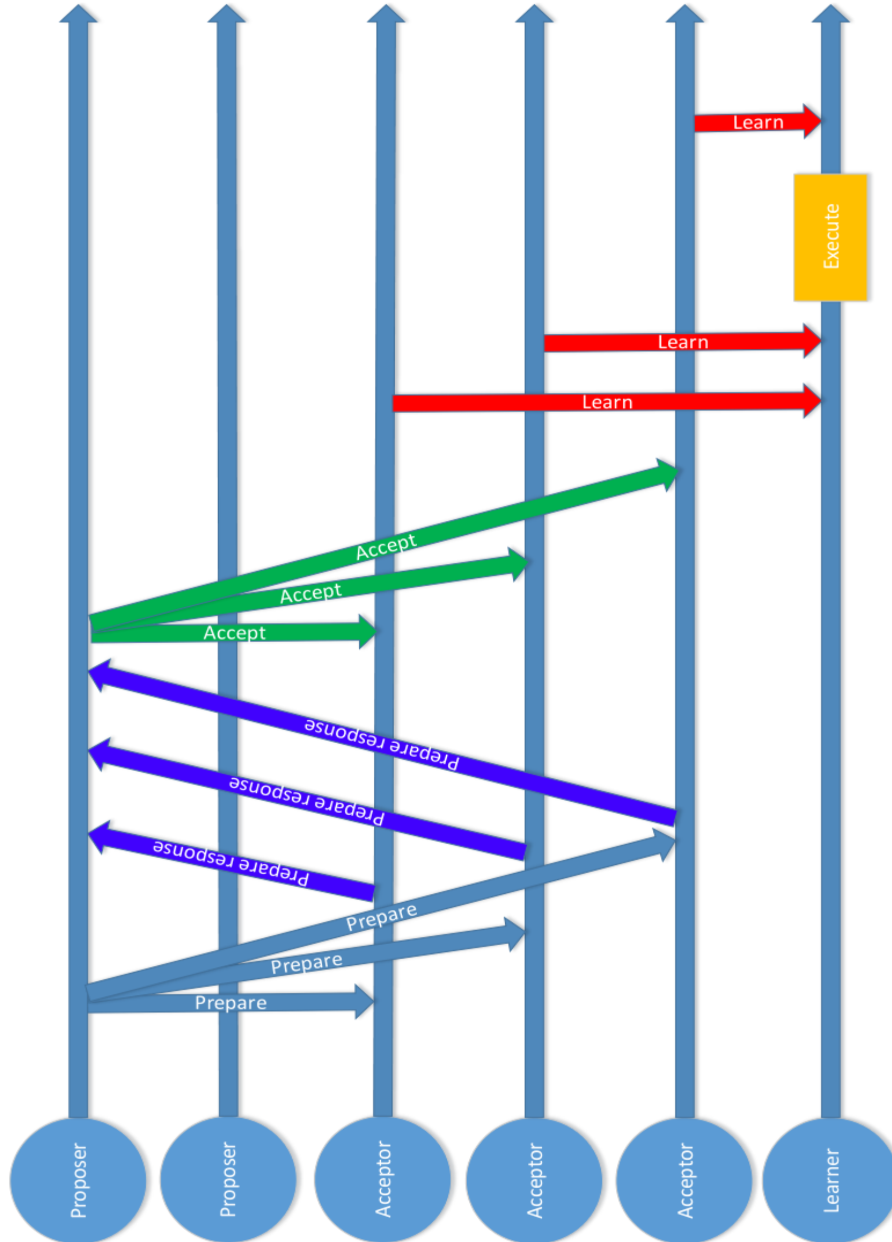
It took a long time to prove that solving consensus in a partially synchronous system is possible [10]. Paxos was one of the first fault tolerant consensus protocols to solve consensus in such a system. It was first proposed in 1989 by Leslie Lamport [15] and explained using a fictional legislative consensus system. The value of Leslie Lamport's protocol stayed unnoticed until 1998 when it was published in a journal. Since then, several variations with different trade-offs were published (e.g. [13, 16, 17]). A participant of the protocol can perform different roles:

- **Proposer**: proposes requests from clients to a quorum of acceptors.
- **Acceptor**: accepts proposals from proposers and can refuse if there are conflicts. Proposers send a proposition to a quorum of acceptors and can only proceed further if the proposition is accepted by a quorum.
- **Learner**: learns the proposed values. A learner is passive and is the replication part of the protocol.
- **Leader**: if there are several proposers, their proposals may conflict indefinitely. To guarantee progress, only a distinguished leader proposes the values. Electing a leader is an optimization.

At the beginning, a proposer sends a *prepare* message with a proposal number N and the value from a client request to a quorum of acceptors. The acceptors accept the proposal and reply with a *prepare response* message, if the proposal number N is larger than any previously seen proposal number. Otherwise, the acceptors do not answer or send a negative-acknowledgement. If multiple proposers propose different values, the *prepare* messages let the acceptors detect conflicts.

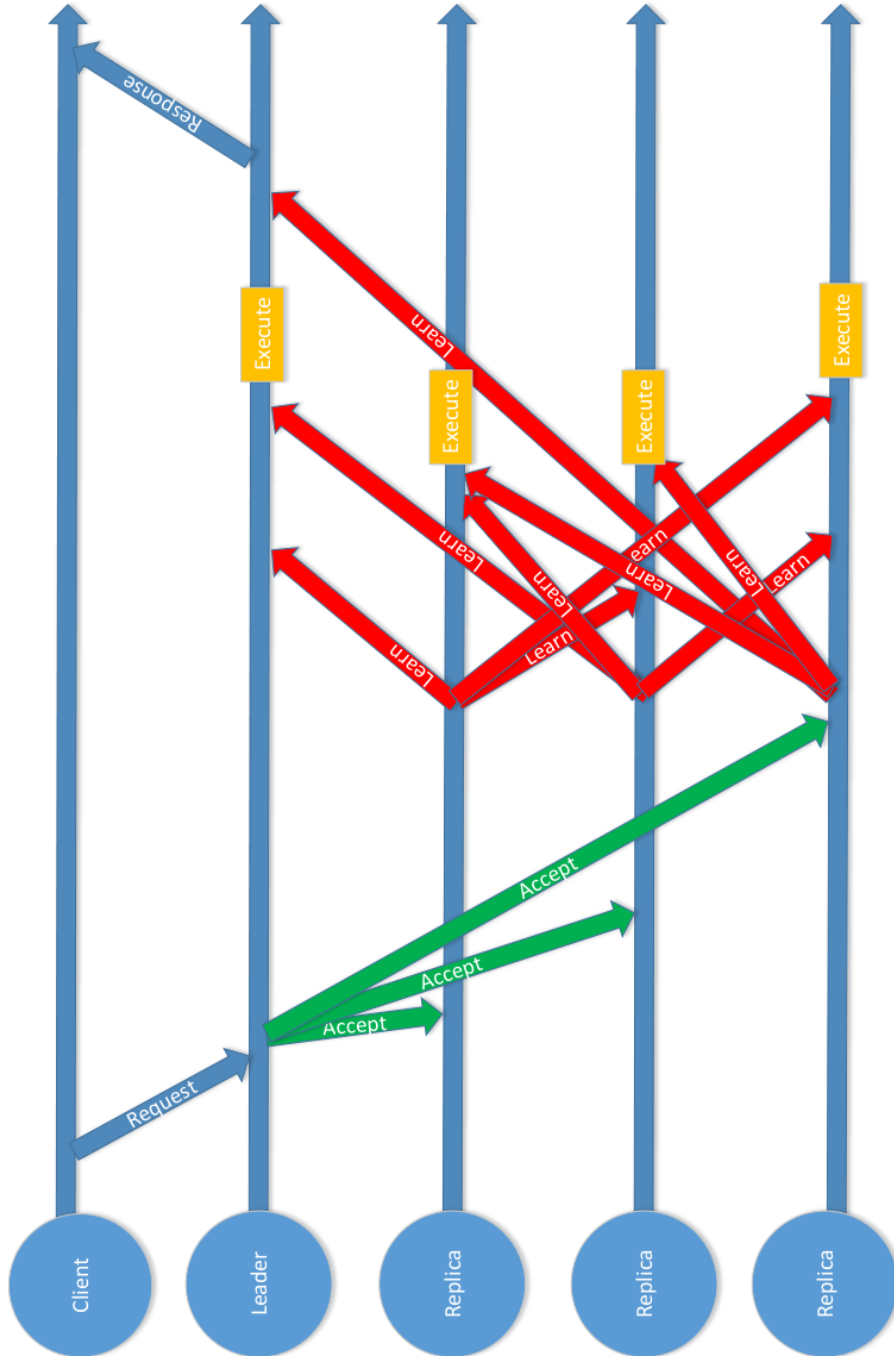
In a second phase, the proposer sends an *accept* message to a quorum of acceptors. The *accept* message contains the same values as a *prepare* message. If the proposal number N is large enough, the agreement succeeds and the acceptor broadcasts a *learn* message to inform the learners of the chosen value. If a learner receives *learn* messages from a quorum of acceptors, it commits the chosen value.

Figure 3.2: Paxos: base case without failure



Paxos only considers a single agreement, but most systems require a stream of agreements. For multiple agreements, Multi-Paxos is used. Multi-Paxos assumes that only a single proposer is chosen as a leader and that the leader is rather stable.

Figure 3.3: Multi-Paxos: base case without failure



When the leader is chosen, the first phase with *prepare* messages is no longer necessary and the number of delays to reach an agreement is reduced. A more common case where the roles are combined (replica is acceptor and learner) is shown in figure 3.3. Through the forming of quorums of acceptors, Paxos as well as Multi-Paxos can tolerate non Byzantine failures. As long as a quorum of acceptors can be formed, Paxos and Multi-Paxos do not stall.

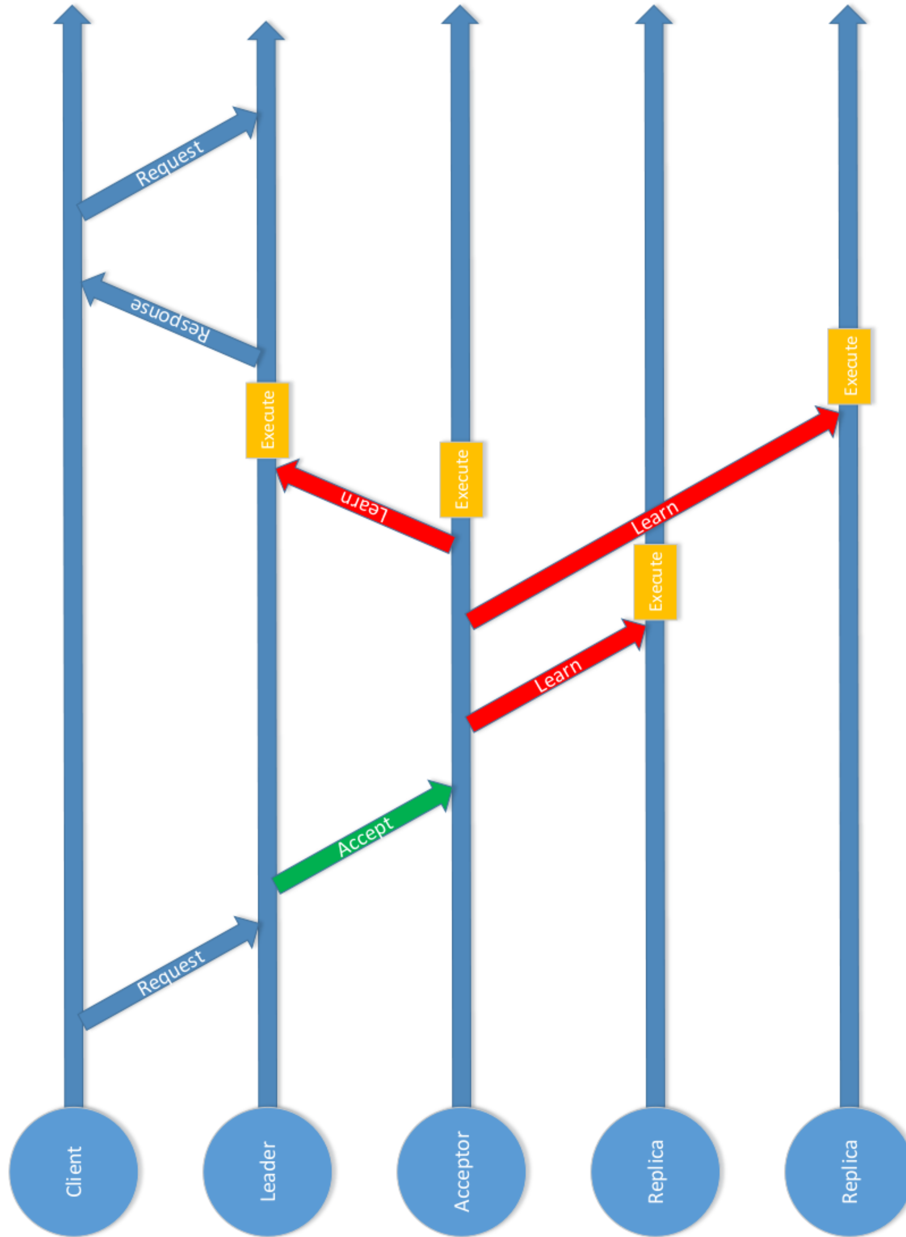
3.3 1Paxos

1Paxos, a Multi-Paxos variant adapted to a multicore machine, was proposed by Rachid Guerraoui et al. [9]. The goal of 1Paxos is to reduce the number of messages per agreement to the minimum. The key concept of 1Paxos is the reduced number of acceptors. If there is only a single acceptor, the number of messages from proposer to acceptors is reduced. Further, since only one acceptor receives an accept, only a single broadcast to the learners is performed. In 1Paxos the message types have the same meaning as in Paxos.

Failure-free case

1Paxos chooses the leader, similar to Multi-Paxos, through *prepare* and *prepare response* messages. A proposer that wants to become leader sends a *prepare* message to the active acceptor. If the proposal number of the *prepare* message is greater than all previous proposal numbers, the acceptor answers with a *prepare response* message. Assuming the leader and acceptor are elected, the leader can send an *accept* message to the acceptor. The acceptor decides if it accepts the proposal and broadcasts the accepted value to the learners depending on the proposal number. The failure free case with an already chosen leader is shown in figure 3.4.

Figure 3.4: 1Paxos: base case without failure



Acceptor Failure

The leader is the only replica authorized to change the active acceptor. The change from one acceptor to another must be confirmed by a majority of replicas to avoid more than one acceptor/leader pair in the system. Agreeing on a new acceptor is achieved by a utility consensus protocol. The leader first ensures that it is still the leader by inquiring all replicas to send it the leader id. If the leader has received a majority of responses with its id, it announces the change of acceptor to the other replicas. Afterwards, the leader sends a *prepare* message to the new acceptor. The *prepare* message contains uncommitted proposed values to cover the case if the acceptor received an *accept* message, but did not broadcast the *learn* message before failing.

Leader Failure

Switching the leader is similar to a failure of the acceptor. A replica can try to become leader if it suspects that the leader is no longer responsive. A replica first acquires the acceptor id by requesting the id from all the replicas. If the replica receives a majority of answers with the same id, it can proceed and announces the change of leadership. The new leader requires a majority of replicas to prevent a change of acceptor during the leader election. After the election, the new leader announces its leadership to all the replicas. If an old leader received a change of leadership message, it assumes that it has been replaced as the leader.

Failure of Both Leader and Acceptor

If both the leader and acceptor fail, the protocol stalls until either the leader or the acceptor is responsive again. The decision to handle this case by waiting is based on the probability. Assuming the probability of two replicas failing at the same time is already small, the probability of exactly the leader and acceptor failing is even less.

3.4 Raft

Raft [23] is a more recent consensus protocol designed to be more understandable than Paxos. Raft decomposes the key elements of consensus such as leader election, log replication, and safety. Further, a strong coherency between replicas is enforced leading to a reduced state space. Raft can tolerate non Byzantine failures as long as majority of replicas is still running. Since the paper [23] contains a lot of implementation details, we explain Raft in more detail.

In Raft replicas can be in one of three states: leader, follower, or candidate. The leader receives requests from clients and sends messages to followers to replicate the state. Followers are passive and just react to incoming messages of the leader. Only if a follower does not get a message in a certain time period, it

becomes a candidate and tries to become the new leader. The leader prevents followers from becoming candidates by periodically sending a so called *Heartbeat* to all followers. On becoming a candidate, the follower increases its term. The term helps to discard messages from old leaders.

State

Hereafter is a summary of the state each role has to keep track of. For all servers there are the following values:

- `currentTerm` **:** the highest term a replica has received in a message. Increases when a follower becomes a candidate.
- `votedFor` **:** the id of the candidate that received the replica's vote during a leader election. Prevents replicas to vote for different candidates in the same term.
- `log[]` **:** a log entry contains a command, which should be applied to the state machine, and the term.
- `lastLogIndex` **:** the index of the last entry that was written to the log.
- `commitIndex` **:** the index of the last log entry that is replicated among a majority of replicas.
- `lastApplied` **:** the index of the last log entry that was applied to the state machine.

Additionally to these values, the leader requires the two arrays:

- `nextIndex[]` **:** the index of the next log entry that the leader can send to a follower.
- `matchIndex[]` **:** the index of the last log entry known to be replicated on a certain follower.

The main mechanisms of the Raft protocol are implemented in two Remote Procedure Calls (RPCs). The RPCs are called *AppendEntry* and *RequestVote*. The *AppendEntries* RPC is executed for both replication and *Heartbeats*.

Remote Procedure Calls

An *AppendEntries* RPC has the following arguments and results:

- **[In]** `term` **:** the leader's term.
- **[In]** `leaderId` **:** the leader's id.
- **[In]** `prevLogIndex` **:** the log index before the new entry. Required for consistency checks.

- **[In]** `prevLogTerm` **:** the term of the previous log entry.
- **[In]** `entries[]` **:** one or several entries that should be appended to the log. Empty for *Heartbeats*.
- **[In]** `leaderCommit` **:** the leader's `commitIndex`.
- **[Out]** `term` **:** the current term.
- **[Out]** `success` **:** result of the RPC's consistency checks.

The entries that are appended to the log have to meet certain conditions. First of all, the `term` must be larger or equal to the follower's `currentTerm`. If the `term` is smaller, an old leader is still trying to append entries. Further, the entry at `prevLogIndex` in the `log[]` must not be empty and match the `prevLogTerm`. If there is a conflict, meaning there is a non empty entry at the location of the new entry that is appended, all the entries following the new entry are deleted. If an *AppendEntries* RPC fails, the leader retries with the log entry before the log entry that failed. In this manner, the leader enforces its log to any follower.

A new leader is elected by performing *RequestVote* RPCs. The arguments and results are:

- **[In]** `term` **:** the candidate's term. Used to check if the candidate's log is at least as up-to-date as the follower's log.
- **[In]** `candidateId` **:** the candidate's id trying to become leader.
- **[In]** `lastLogIndex` **:** the last index added to the log. Used for up-to-date log check.
- **[In]** `lastLogTerm` **:** the term of the last log index.
- **[Out]** `term` **:** the current term of the follower.
- **[Out]** `voteGranted` **:** if the vote is granted.

If a candidate receives a majority of votes from followers, it becomes the new leader. If there are two candidates trying to become leader and they tie, they initiate another vote. The time a follower waits until it initiates a vote and becomes a candidate, is the sum of the base wait time and a randomized wait time. The randomization minimizes the number of clashes between two candidates and a new leader is elected faster.

Leader Election

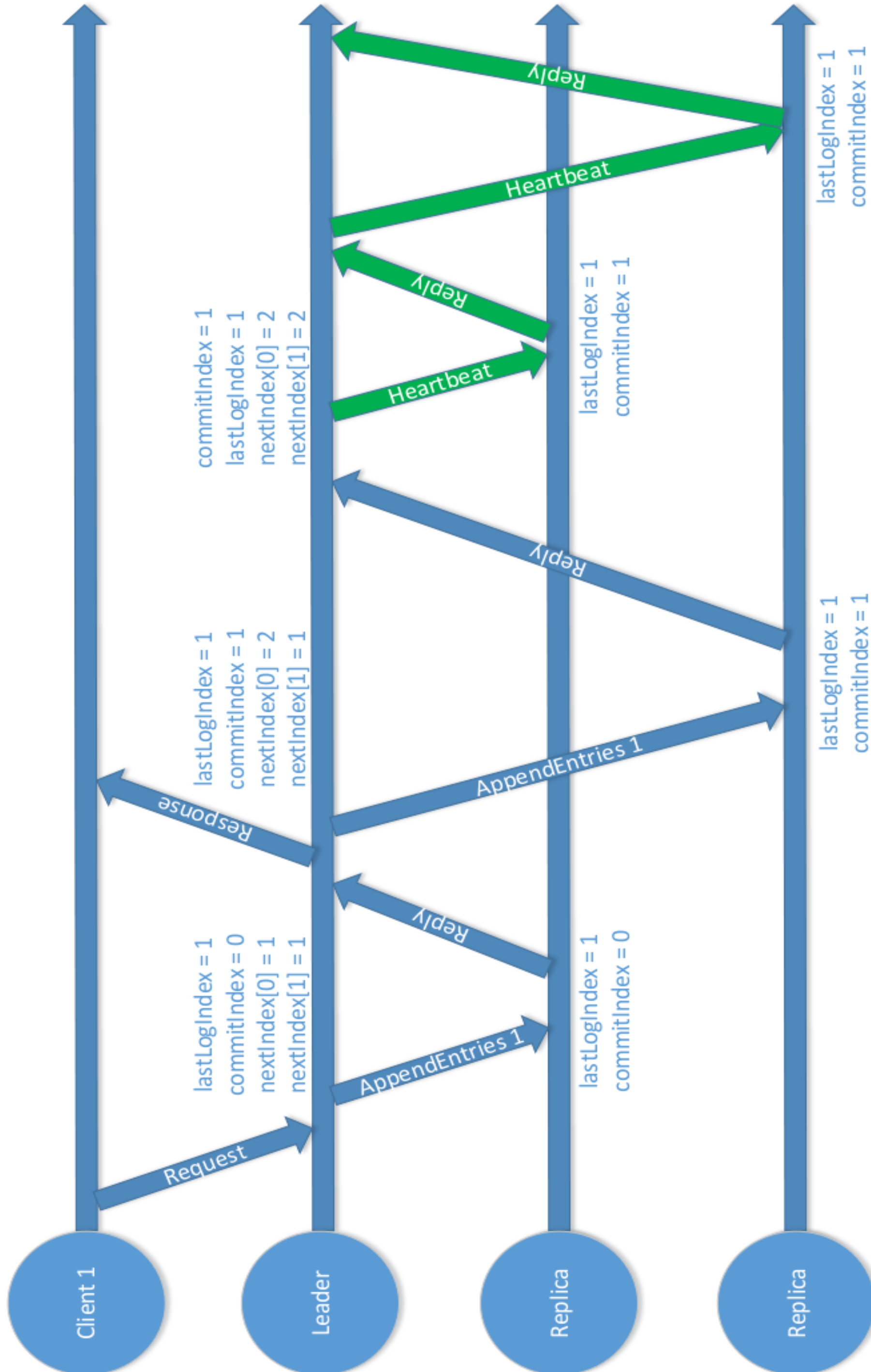
When a replica becomes leader, the replica periodically executes an RPC called *Heartbeat*. A *Heartbeat* is an empty *AppendEntries* RPC. The *Heartbeat* RPC prevents followers from converting to a candidate. As long as a follower receives messages from either a candidate or a leader, it stays in the follower state. If

a follower becomes a candidate it increases its term, votes for itself, and starts to request votes from other replicas. A candidate is only elected if it receives a majority of votes. To prevent candidates with a less up-to-date log to become leader, a candidate only receives positive votes if it has a more up-to-date log than the follower it requests the vote from.

Log Replication

If the leader receives a request from a client, it writes the request down into the log. The leader replicates the log by performing an *AppendEntries* RPC to each follower. Further, each follower has two values: the `commitIndex` and `lastApplied`. An entry in the log is committed if the leader has replicated the entry on a majority of followers. With each *AppendEntries* RPC the leader sends its `commitIndex`. If the `commitIndex` of the follower is less than the `commitIndex` of the leader, the follower increases its `commitIndex`. The leader has two arrays that it has to keep track of: `nextIndex[]` and `matchIndex[]`. The `nextIndex[]` keeps track of which is the next log entry the leader can send to a certain follower. If `lastLogIndex` $\geq$ `nextIndex[]` holds for a follower, the leader tries to send further entries. The `matchIndex[]` is the highest log entry that is replicated on a follower. The leader can increase its `commitIndex` to N, if a majority of `matchIndex[]` is larger than N and the `term` of this log entry matches the current term. The `lastApplied` value is the last entry that is applied to the state machine. If the `commitIndex` of a follower is less than `lastApplied`, the follower increases `lastApplied` and applies the log entry at the index `lastApplied` to the state machine. The base case of Raft is shown in figure 3.5.

Figure 3.5: Raft: base case without failure



3.5 Evaluation

Out of a multitude of consensus algorithms we decided on the following protocols: 2PC, Raft, and 1Paxos. 2PC is our base line and simple to implement. Since 2PC does not handle failures, it should have a better performance than any algorithm handling failures. We consider 2PC to be a valid option to run within NUMA nodes. Some years ago, a Multi-Paxos implementation was the way to solve consensus, but within a multicore environment Multi-Paxos is a poor choice because of the excessive use of messages. Raft is a protocol that does not require a lot of messages and may fit into a multicore environment. 1Paxos is the last protocol we wanted to implement since it is developed to fit into a multicore environment. 1Paxos utilizes the facts that the protocol is running on a multicore machine and makes full use of the added guarantees.

Chapter 4

Improvement for Multicore

Before we can address the issues with consensus in a multicore environment, we have to understand the problems. At the moment, there are a few framework that consider consensus on a multicore machine. The framework most similar to our goal is from Rachid Guerraoui et al. [9]. Luckily for us, the code is publicly available. The code contains an implementation of 1Paxos, 2PC, and Multi-Paxos. This seems to be a good foundation for further analysing the requirements for running consensus on a multicore machine.

4.1 Consensus Inside

The framework of Rachid Guerraoui et al. [9] contains both the possibility to measure throughput and response time. Three programs are available: client, replica, and manager. A replica participates in the protocols and receives requests from clients. The manager synchronizes the clients. Only after all clients received a message from the manager, they can start sending requests to the replicas.

Message passing between cores is implemented by a framework called QC-libtask. Similar to Barrelfish [24], QC-libtask uses shared memory and exploits the cache coherency protocol to send messages. A communication channel consists of two shared memory queues (send and receive) and is allocated between a pair of cores. Further, a user-level thread is started that polls for new incoming messages. To improve message delivery times, they made use of a user-level thread library that reduces the delivery time to a simple lightweight user-level context switch. The shared memory channels are abstracted by file descriptors. The abstraction leads to two functions for messages passing: `fdread()` and `fdwrite`. `Fdread()` is a blocking call and may send the receiving thread of the channel to sleep.

4.2 Micro Benchmarks

Micro benchmarks are a helpful tool to analyse issues. Fine grained measurements help to pinpoint the code segments where time is lost.

4.2.1 Setup

For our experiments we made use of a 4×12 AMD Opteron(tm) 6174 processors at 2.2 GHz with a total of 128 GB RAM. The CPUs are not native 12 cores but actually 2×6 cores with a shared L3 Cache. The machine runs on Ubuntu 12.04.

The machine is divided into 8 NUMA nodes with 6 cores per NUMA node. The cores are assigned to NUMA nodes according to table 4.1.

Table 4.1: Micro benchmarks: cores assigned to NUMA nodes

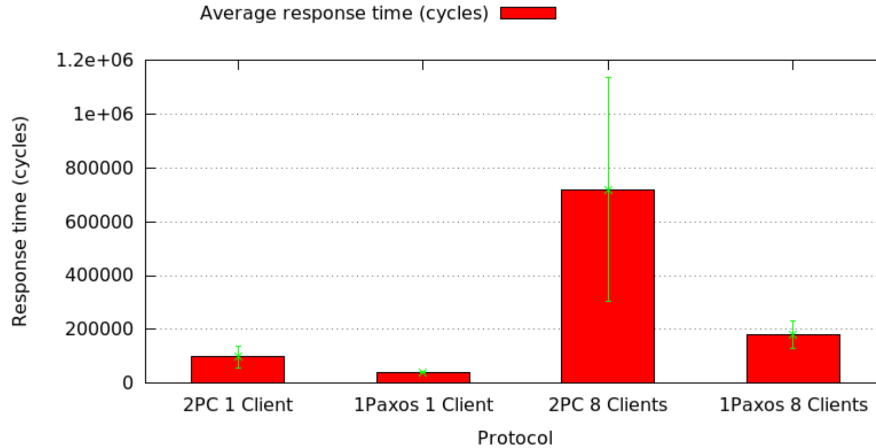
Node Number	Cores
Node 0	0,4,8,12,16,20
Node 1	24,28,32,36,40,44
Node 2	1,5,9,13,17,21
Node 3	25,29,33,37,41,45
Node 4	2,6,10,14,18,22
Node 5	26,30,34,38,42,46
Node 6	3,7,11,15,19,23
Node 7	27,31,35,39,43,47

The programs we mentioned before (client, replica, manager) are assigned to cores by the `taskset` command. The manager is always running on the last core. The replicas are assigned to the first core on each NUMA node. Every other core is used for clients. The assignment avoids putting clients and replicas on the same core. When a client and a replica run on the same core, there is a large overhead for context switching. For each request there would be a kernel context switch from client to replica because sending a request is blocking.

4.2.2 Response Time

The response time is a crucial indicator for the usability of the consensus protocols in context of an agreement service within an operating system. For an operating system 10'000 cycles are already a long duration. We measured using 8 replicas and put load on the system with 1 or 8 clients to see the behaviour for different loads.

Figure 4.1: Response time: comparing 1Paxos and 2PC using 8 replicas



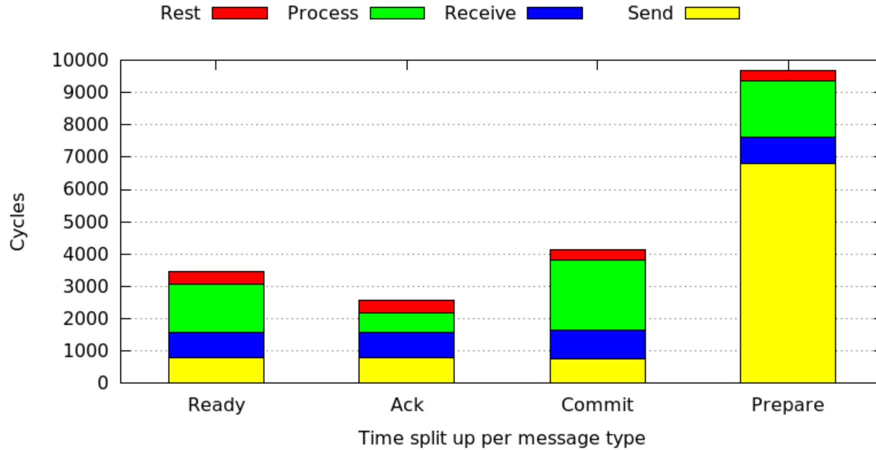
1Paxos has a response time of 42'000 cycles where as 2PC is already around 98'000 cycles for a single client. A few tens of thousands cycles are too much let alone when there is a higher load with 8 clients (180'000 and 720'000 cycles). 1Paxos has a reasonable standard deviation where as the standard deviation of 2PC gets out of hand. To pinpoint where the time is lost, we did further benchmarking but on a finer granularity.

4.2.3 Message Processing

We added some code to the existing benchmark implementation that measures the time in different code segments and prints the results into a file at the end of the benchmark. For each protocol there are different types of messages that take a different amount of time to process. The code is split up into measurement segments of *sending*, *receiving*, *processing*, and *rest*. The *sending* part is an invocation of `fdwrite()` that writes into the shared memory region. The *receiving* part is more complicated. QC-libtask starts a receiving thread for each channel it allocates. The receiving thread may also be sent to sleep. The receiving time only accounts for the time it takes to receive the data and not the sleeping/waiting time. The receiving time includes the time to copy a message from the queue into some allocated memory. The *processing* time accounts for the logic of the consensus protocol and the *rest* is the time the measuring code takes up.

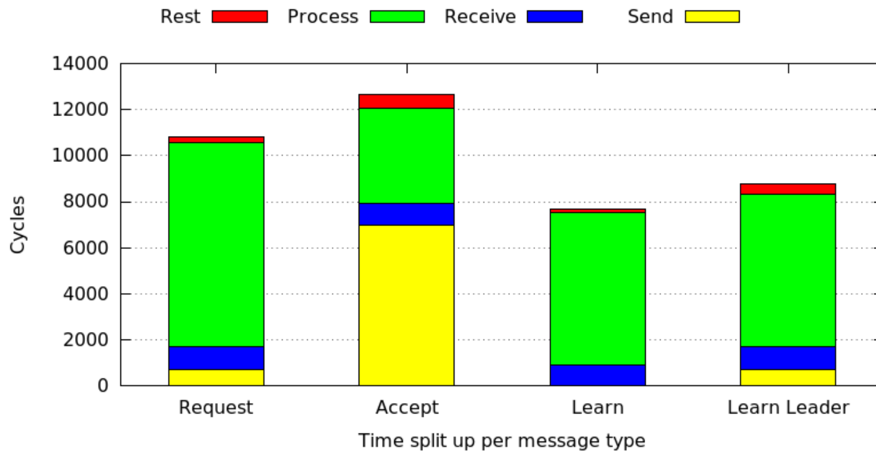
The replicas are started on cores 0-3 and 24-27. The benchmark consists of a single client sending 1'000 requests. We benchmarked both 2PC and 1Paxos. Note that figure 4.2 shows the time from receiving a certain message type including the reaction to this message type.

Figure 4.2: Micro benchmark: 2PC message types using 8 replicas



2PC has four message types: *prepare*, *ready*, *commit*, and *ack*. The only broadcast that is executed for each received message, is from the leader to the other replicas with the *prepare* message type. The *commit* message type involves only a single broadcast after receiving a *ready* message from all other replicas. We amortized the cost of the send operation for the *commit* message type. Our assumptions are that the send and receive time should dominate the cost. The *ack* and *prepare* message types uphold the assumption. The *ready* and *commit* message types are dominated by the processing cost.

Figure 4.3: Micro benchmark: 1Paxos message types using 8 replicas



For 1Paxos we excluded the message types *prepare* and *prepare response* since they are only sent once at the beginning. We have two different measurements

for the *learn* message type since the leader has to react differently. The results are shown in figure 4.3.

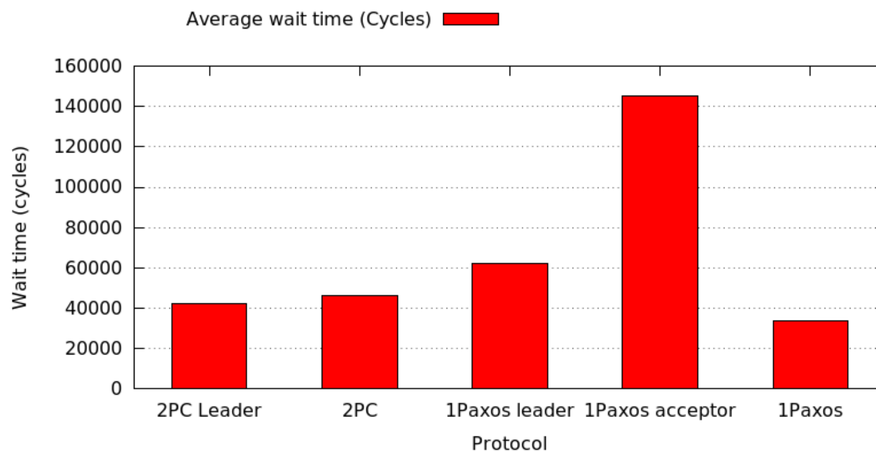
The three message types of 1Paxos are: *request*, *accept*, and *learn*. 1Paxos in the base case only requires a single sequential broadcast of a *learn* message from the acceptor to all other replicas. If the leader receives the *learn* message, it informs the client that the agreement is completed. The receiving and sending time matches the 2PC results, but the time spent processing the messages is high. The processing cost clearly dominates if there is no sequential broadcast.

Just comparing the total time spent per message type 2PC seems to be faster than 1Paxos. Considering the logic of the protocols, it is clear why 2PC has a longer response time. 2PC has to wait for all replicas to respond. 1Paxos only waits for the leader to receive a *learn* message. Still, there are large discrepancies between the response time and the sum of the time spent per message type. There is one quantity left that we did not take into account: the wait time from receiving a message.

4.2.4 Wait Time

The wait time is the time `fdread()` blocks until data arrives and the thread gets CPU time to process. Since QC-libtask starts a receiving thread for each channel, there is a lot of potential overhead from switching between threads and queueing threads for CPU time. The wait time can not be added directly to the values measured before. A receiving thread may wait for an incoming message since there is nothing else to process. Doubtless, the wait time is accountable to some part of the response time. We excluded the wait time from the previous graphs out of readability reasons since the values are large.

Figure 4.4: Micro benchmark: wait time using 8 replicas

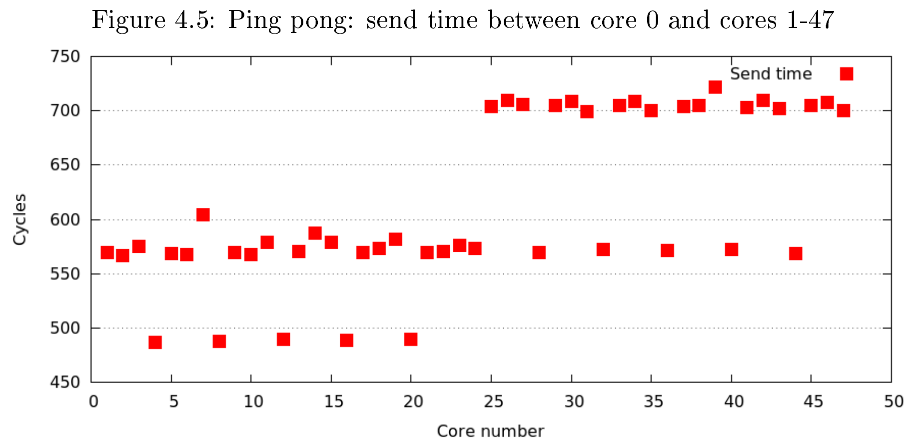


The wait time is hard to decipher. The acceptor and leader of 1Paxos exhibit a higher wait time than 2PC. A higher wait time can indicate both high load or low load. If the load is high, the wait time increases because several threads are queued and wait for CPU time. If the load is low, the threads may not have anything to process and the wait time increases. Still, the high wait time indicates that one thread per receiving channel is too much.

4.2.5 Ping Pong Benchmark

In a last benchmark, we are interested in how the send time changes depending on which two cores are communicating. If there is a difference we can optimize the communication patterns between cores. We wrote a benchmark that does a ping pong between two cores on top of QC-libtask. One of the cores starts sending a single message and both cores react to receiving a message by responding back. We pinned one of the programs to core 0 and the other program changes from cores 1 to 47. For each core we did three measurements with 1'000'000 messages sent back and forth per measurement.

Figure 4.5: Ping pong: send time between core 0 and cores 1-47



The send time clearly shows the NUMA topology. The communication between core 0 and cores 4,8,12,16, and 20 that form the NUMA node 0 is cheaper than with any other core. Since the AMD CPU is actually two 6 cores CPUs on the same die, the cores of NUMA node 1 (24, 28, 32, 36, 40, 44) show a similar effect. The cores of nodes 2 (1, 5, 9, 13, 17, 21), 4 (2, 6, 10, 14, 18, 22), and 6 (3, 7, 11, 15, 19, 23) seem to have a direct communication channel with node 0. Any other core shows a higher number of cycles and we can assume that there is a further hop over another NUMA node necessary. The difference from the cheapest communication path to the most expensive is around 250 cycles.

Pinpointing the program on core 0 to another core yields other results but shows the same NUMA topology. Changing the core only shows which cores

belong to a local node, have a single hop or more than one hop to the next node.

4.3 Tree Broadcast

From subsection 4.2.3 we can infer that a sequential broadcast is a costly operation. Instead of letting a single replica execute the broadcast, the replicas can broadcast along a one-sided tree as suggested by Nishtala and Yelick [21]. If the broadcast is along a one-sided tree, the cost is not centred at a single replica but distributed to some degree to other replicas. An example of a one-sided broadcast tree is shown in figure 4.6.

Figure 4.6: Tree broadcast: example of a one-sided tree

The root of the tree first sends a message to its leftmost child. While the child has already received the message, the root sends the next message to the second child. When we assume the propagation time is almost non existent, the root finishes the broadcast when the leftmost node of the tree receives the message. The colors in figure 4.6 represent rounds. In a first round the root sends to the red node. In the next rounds the blue nodes receive a message, then green and at the end the orange nodes. By broadcasting along a tree, the root only has to send messages to 4 nodes instead of 15. The load on the root is reduced and broadcasts are more scalable to a higher number of nodes.

A drawback of a broadcast tree is failure handling. If one of the nodes within the tree fails, the whole sub-tree does not receive the broadcast. The parent of the failed tree node has to detect the failure and replace the failed node. We did not consider the broadcast tree for consensus protocols that can handle failures because of the added complexity.

4.4 Protocol Implementation Overhead

In our opinion, the cost of a consensus protocol running on a multicore machine should be dominated by send and receive operations. In subsection 4.2.3 we showed that the processing cost of the messages dominate. When we had a closer look at the code (written in C++), we discovered several problems. Since C++ supports object orientation, the message types are represented as classes. When a message is received, there is an overhead for converting from one object type to another type. Converting the message type often involved `malloc()` or `memcpy()`. There are simpler solutions to change the type of a message. The cost of `malloc()` and `memcpy()` depending on the size can vary from a few hundred to thousands of cycles.

Further, several C++ library functions are used that can be replaced with simpler primitives. The more we rely on library functions, the less we can control what our code does. C++ may deliver features that simplify the implementation of complex consensus protocols, but in the context of an operating system C is still the better choice.

4.5 Selection Overhead

QC-libtask starts a receiving thread for each channel it sets up. As an example, if a leader has to send messages to 7 other replicas (8 replicas total) there are 7 user-level threads running and competing for CPU time. The matter gets worse if we account for clients that send requests to the leader. A thread per client or replica does not scale.

Let us have a look at the composition of the response time of 1Paxos. For the moment let the time between sending on one core and receiving the message on the other core be zero (almost the case). A client sends a request to the leader where it is forwarded as an *accept* message to the acceptor after around 10'800 cycles. The acceptor requires another 12'600 cycles to broadcast the *learn* messages, but the message to the leader is sent first after around 6'000 cycles. On receiving the *learn* message the leader requires another 8'800 cycles to send the reply to the client. The sum accounts for 25'600 cycles of the total 42'000 cycles response time. Selecting and running the required receiving threads should account for nearly all the rest of the missing cycles.

Barrelfish has a solution to mitigate the selection problem: waitsets and event handling. A waitset contains three queues for channels: *idle*, *pending*, and *polled*. If a channel raises an event like the possibility to receive a message, it is put into the pending queue. An application can call `event_dispatch()` that removes the channel from the pending queue and returns the event (function pointer and arguments) and executes it. In this manner, one thread to dispatch events per replica is enough.

Chapter 5

Framework Design

The goal of this thesis is to establish an agreement framework for Barrelfish. Composability had a central role in the framework design and helps both performance and failure tolerance. A uniform client and replica interface helps to compose protocols. An option to automatically compose protocols according to a policy delivers optimal performance for the underlying multicore machine.

5.1 Composability

One size fits all can not be applied to a multicore machine. A multicore machine is a two level hierarchy (within NUMA nodes, between NUMA nodes). Out of this reason, we came up with the idea of a framework that composes protocols on different levels of the hierarchy. The composition works from top to bottom with the higher layer starting the lower layers.

For the beginning there are only two levels in the hierarchy. In the future we may add a higher layer to run protocols between machines. A composition has advantages for both performance and failure tolerance.

5.1.1 Performance

In computer science divide and conquer is a well known concept. Composing solutions for a smaller part of the system to find a solution for the whole system is a reasonable approach. Often, a consensus protocol relies on a leader or at least contains some form of broadcast to all replicas. In both cases, a single replica is burdened with the cost of communicating to all other replicas. Sending the messages along a tree can reduce the load but handling failures is hard. The problem of high load on certain replicas can also be solved by a composition. A hierarchy of protocols represents a tree. Similar to a tree, the number of replicas the leader communicates with is reduced. Even just composing the same protocol on two layers should have a better performance because the load

is more evenly distributed to different cores.

We considered two layers: within a NUMA node and between NUMA nodes. The hierarchy gives us the advantage that we can compose hybrid combinations of message passing and shared memory. Consensus protocols are often hard to scale to a large number of replicas. Combining protocols using message passing (between nodes) and shared memory (within nodes) should give a good performance boost and enable us to scale to a larger number of replicas.

5.1.2 Failure

In section 2.3 we defined our failure model and failure domain. Within a failure domain we can run protocols that do not have to consider failures. In theory, a protocol that does not have to deal with failures should always yield a better performance than a protocol that has to deal with failures. Moreover, within a failure domain shared memory is an option that should not be discarded.

Between failure domains we have to rely on a protocol that can handle failures. The cost of tolerating failures can be high, therefore only a single replica within a failure domain should participate in the higher layer protocol.

5.2 Interfaces

A common interface is the tool to get the composability to work. In the current code there are four interfaces: an internal interface for replicas to start the lower layer, the replica interface, the client interface, and an interface to start the service.

5.2.1 Replica Interface

A user needs to define what happens when the replicas have agreed on a value. There is no dynamic linker for Barrelfish yet and consequently, a user has to provide a program that starts a replica. In this manner, the user is able to set the function that should be executed when an agreement is finished. The interface to start a replica is shown in listing 5.1.

Listing 5.1: Composability: replica interface

```
#define CORE_LEVEL 0
#define NODE_LEVEL 1
...

#define ALG_1PAXOS 0
...
#define ALG_NONE 6

void init_consensus_replica(int algo,
                            int id,
```

```
                                   int num_clients,
                                   int num_replicas,
                                   int level,
                                   int alg_below,
                                   int node_size,
                                   int started_from,
                                   uint16_t* cores,
                                   void (*exec_fn)(void *),
                                   char* prog_string);

void init_consensus_replica_argv(char** argv,
                                 void (*exec_fn)(void *));

void set_execution_fn(void (*exec_fn)(void *));
void start_handler_loop(void);
```

The most complex function is `init_consensus_replica()`. It initializes a replica by setting up the internal data structures and opening the communication channels to all other replicas. In our example replica programs, all of the arguments for the replica initialization are parsed from the main functions `argv` array. The arguments have the following meaning:

- `algo:` the algorithm or protocol that the replica should run. For each algorithm/protocol we defined an integer value.
- `id:` the id of the replica is unique, but only within a level of the hierarchy.
- `num_clients:` only for benchmarking purposes.
- `num_replicas:` the number of replicas started on this level.
- `level:` the level on which a replica is started. At the moment only `CORE_LEVEL` and `NODE_LEVEL`.
- `alg_below:` the algorithm or protocol that should be started on the layer below.
- `node_size:` the size of a NUMA node (or cluster).
- `started_from:` since several instances of a protocol are running on the lower level, we need to differentiate between them by the replica id of the higher layer.
- `cores:` the cores of the NUMA node (or cluster) excluding the core of the higher level replica. If a NUMA node contains the cores 0-5 the cores array contains the numbers 1-5.
- `exec_fn:` the function that should be executed after an agreement.
- `prog_string:` the string of the program that starts a replica. Used to start replicas on the lower layer.

Since a user often may not be interested in any values given to init_consensus_replica(), we also provide the function init_consensus_replica_argv() that takes the argv string array from the main function and parses all the required arguments from it.

After a replica is initialized, it can start handling messages by starting the blocking message handler loop (start_handler_loop()). We split up the functionality of initializing and handling messages to provide the user a possibility to start a client in between. A client can only be started after the replicas are up and running. Otherwise, the client initialization blocks. For best performance while starting both a client and a replica on the same core, they should be started in the same domain. This prevents a kernel context switch per request and leaves it at a much cheaper user-level thread context switch. Still, the best way to handle client and replica core allocation is by putting the two applications onto separate cores.

5.2.2 Internal Interface

The internal interface is not shown to users and abstracts the lower layer to two functions. One function starts the lower layer within a NUMA node and the other sends requests to the lower layer. The interface is shown in listing 5.2.

Listing 5.2: Composability: lower layer interface

```
#define ALG_1PAXOS 0
...

void com_layer_core_init(uint16_t algorithm,
                         char* replica_string,
                         uint16_t replica_id,
                         uint16_t* cores,
                         uint16_t num_cores,
                         uint16_t cmd_size);

void com_layer_core_send_request(void* addr);
```

The function com_layer_core_init() initializes a protocol within a NUMA node. It is normally called in the initialization of a higher layer replica. The arguments of com_layer_core_init() have the following meaning:

- **algorithm:** the value decides which algorithm or protocol is started within a NUMA node.
- **replica_string:** the string of the replica program.
- **replica_id:** the id of the higher layer replica. Distinguishes the protocol instances on the lower layer.
- **cores:** each element of this array represents a core on which a replica should be started.

- `num_cores`: the number of cores that are in the array `cores`.
- `cmd_size`: the size of the payload (only used for shared memory).

The function to send a request has a void pointer as an argument. Internally the payload of a protocol is defined in the Flounder interface. As a consequence each protocol has a differently named struct. To send the payload further down to the lower layer, we simply cast these structs to a void pointer.

5.2.3 Client Interface

The client side of the framework does not require a lot to be able to send a request to the service. A client needs a function to initialize itself as well as a function to send requests with a fixed size payload to the service.

Listing 5.3: Composability: client interface

```
struct consensus_payload {
        uint64_t arg1;
        uint64_t arg2;
        uint64_t arg3;
};

errval_t init_consensus_client(void);
errval_t consensus_send_request(struct consensus_payload* payload);
```

The struct `consensus_payload` is limited to three `uint64_t` values to avoid letting messages grow to sizes larger than a cache line (64 bytes). The usage of the client side of the agreement service is simple. First, call `init_consensus_client()`. Second, assemble the payload and send a request to the service by calling `consensus_send_request()`.

A client automatically detects which consensus protocol is running and acts accordingly. A client can only detect the protocol through a uniform Flounder interface that every replica has to implement. The uniform interface for the communication between client and replica is shown in listing 5.4.

Listing 5.4: Composability: replica to client communication interface

```
interface consensus "Interface for Consensus service" {
  typedef struct {
      uint64 arg1;
      uint64 arg2;
      uint64 arg3;
  } command;

  message request(uint16 client_id,
                  uint64 request_id,
                  command cmd);
  message reply();

  message new_leader(uint16 new_leader,
```

```
                    uint64 next_rid);

  message setup(int32 id);
  message setup_response(uint16 client_id,
                         uint16 replica_id,
                         uint16 num_replicas,
                         uint16 algo);
};
```

The struct `command` is the same as the `consensus_payload`. There are two pairs of message types: *request* and *reply*, as well as *setup* and *setup_response*. The message type *request* is sent to a replica to start an agreement and *reply* is received when the agreement ends. With the *setup* message type a client requests the information it needs from the replica with id 0: the client's id and the algorithm/protocol that is running. Further, a client needs to know the number of replicas to set up a connection to every replica. All these additional connections are only needed if the leader changes. The channels are preallocated and the new leader can directly inform clients by sending *new_leader* messages.

5.2.4 Framework Interface

The framework interface contains two functions. The first one lets the user enter the setup and starts the hierarchy accordingly. The second function tries to find an optimal composition and starts it. The functions are shown in listing 5.5.

Listing 5.5: Composability: framework interface

```
void consensus_init(uint16_t algorithm,
                    char* replica_string,
                    uint16_t* cores,
                    uint16_t num_cores,
                    uint16_t num_clients,
                    uint16_t alg_below,
                    uint16_t node_size,
                    uint16_t* node_cores);

void consensus_init_auto(char* replica_string,
                         bool failures,
                         bool full_replication);
```

The arguments to `consensus_init` have a similar meaning to the ones used to start a replica. There is only the difference of the arguments `cores` and `node_cores`. `Cores` is an array of core numbers on which the higher level protocol is started. The `node_cores` on the other hand is an array of size number of nodes $\times$ node size that contains all the core numbers on which the lower layer is started (example: two nodes with cores from 0-3 and 4-7. `node_cores` looks like 1, 2, 3, 0, 5, 6 , 7, 0 while `cores` is 0, 4). The zeros in the `node_cores` array are padding.

The function `consensus_init_auto()` requires the program string of the replicas, if failures should be expected (failure domain: NUMA node) and if the

composition should replicate on all cores or only between NUMA nodes.

5.3 Policy

We divided the policy from the mechanisms as much as possible. There are two decisions to make: when to use which protocol and when to use shared memory.

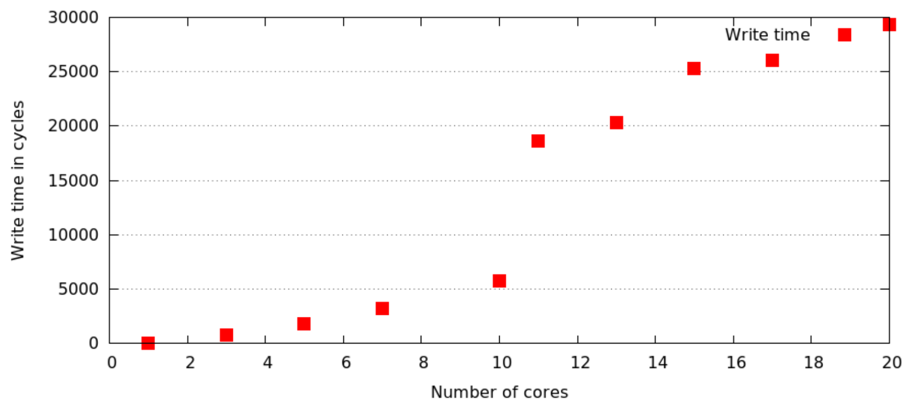
5.3.1 Protocol Choice

We implemented the protocols 2PC, 2PC without leader, 1Paxos as well as simply broadcasting from a leader. If failures are considered, 1Paxos is the obvious choice. If we do not expect failures to happen and the weaker consistency model of causal order is enough, 2PC without leader should offer a higher throughput than 1Paxos. If we do not consider failures, sequentializing the requests and broadcasting them from a leader may be the best choice for performance.

5.3.2 Shared Memory

Within a NUMA node shared memory delivers good performance. Most of the time, the memory is in the L3 cache. We did an experiment to show the difference between sharing memory within a NUMA node and between nodes. We allocated a shared memory region and let several cores write into it. To avoid race conditions, we added a spinlock at the beginning of the shared memory. Since every core needs to access the spinlock, the cache coherency protocol is most likely flooding the bus. The experiment is using a 20 core machine (Intel Xeon E5-2670 v2 2.5GHz 2×10). The cores try to write into the shared memory with no sleep time in between writes.

Figure 5.1: Shared memory: performance of writes w.r.t. number of cores



As long as we share memory within a node, the time it takes to write into the shared memory is dominated by queueing for taking the lock. As soon as we

share the memory across NUMA nodes (11 cores), the time to write into the shared memory jumps up. As part of our policy, we try to avoid writing into shared memory from different NUMA nodes.

5.3.3 Summary

Our policy is a combination of the results of the previous two subsections. If there is only a single NUMA node, shared memory should be enough for replication. If there are several NUMA nodes, we have to choose a protocol between NUMA nodes depending on the failure model. If we tolerate failures, 1Paxos is the obvious choice. If there are no failures we choose broadcast replication. 2PC without a leader may be an option for high throughput, but can only guarantee causal order. We did not consider 2PC without a leader for our policy.

Chapter 6

Implementation

We first implemented the consensus protocols of the framework. In this chapter we focus on the problems that came up when we adapted the protocols to a multicore environment. More implementation details are described in the appendix. After implementing the building blocks, we implemented the layer for composing protocols and an example of how to use the framework.

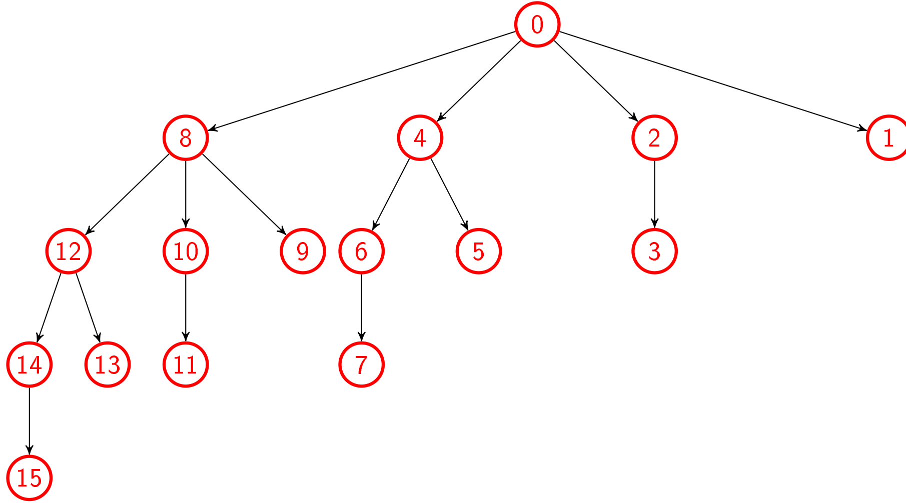
6.1 2PC

The first protocol we implemented is 2PC based on a leader. On a single machine, the communication channels have two further guarantees that a network does not provide out of the box. On one hand, there is the First in First out (FIFO) property that prevents messages from overtaking each other. On the other hand messages can not be lost or dropped. We made use of the second additional guarantee and saved the last *ack* message to the leader. The leader can directly reply to the client after it sent the *commit* messages. Further, we implemented the tree broadcast we explained in section 4.3.

6.1.1 Tree Broadcast

A replica knows where to forward the broadcast by sending along a one-sided radix 2 k-nomial tree [21]. On a radix 2 k-nomial tree each node has a fixed number that we can map to a replica id. Since the whole structure of the tree is static, every replica knows where to forward a received message. An example of assignments of the replica ids to the tree nodes is shown in figure 6.1.

Figure 6.1: Tree broadcast: example of a one-sided radix 2 k-nomial tree



The formulas to construct the tree differentiate between the property of a replica id.

- `Id == 0`: the root sends to all nodes with an id that is a power of two and less than the size of the tree i.e.

$$2^x \text{ where } x = 0, 1, 2 \dots \text{ and } x < \log(\text{size})$$

- `Id == power of two`: the nodes with an id that is a power of two send to all nodes that have a higher id and that are the sum of its own id and a power of two that is smaller or equal than its own id divided by two.

$$\text{id } = \text{my id } + 2^x \text{ where } x = 1, 2, 3 \dots \text{ and } x < \log\left(\frac{\text{my id}}{2}\right)$$

- `Id % 2 == 0`: if the id of the node is even, the formula for the ids to send to is similar to the previous one.

$$\text{id} = \text{my id } + 2^x \text{ where } x = 1, 2, 3 \dots \text{ and } x < \log\left(\text{my id} - \text{parent id}\right)$$

- `Id % 2 == 1`: if the id of the node is odd, there is nothing to do.

Since the structure of the tree is static, every replica can be the root. The only requirement is that the message contains the `sender_id` of this replica. If the `sender_id` is not equal to 0, every other replica can assume that the replica that started the broadcast switches place with the replica with id 0. If a replica needs to send to the replica with `sender_id`, it simply sends to replica with id 0. If the replica with id 0 receives a message with a `sender_id` unequal to 0, it just assumes the role of the replica with the `sender_id`.

6.1.2 Removing the Leader

Since a single leader may limit the performance, we implemented a second version of 2PC without a leader. The clients directly broadcast their requests to the replicas. Conflicts of agreements are detected by a key on which the agreement takes place. The order of the values agreed on is no longer a total order but a causal order. The order of agreements on a certain key are seen the same on all replicas. We hoped to improve the throughput of 2PC by removing the leader since the number of messages a replica has to process is smaller.

6.2 Raft

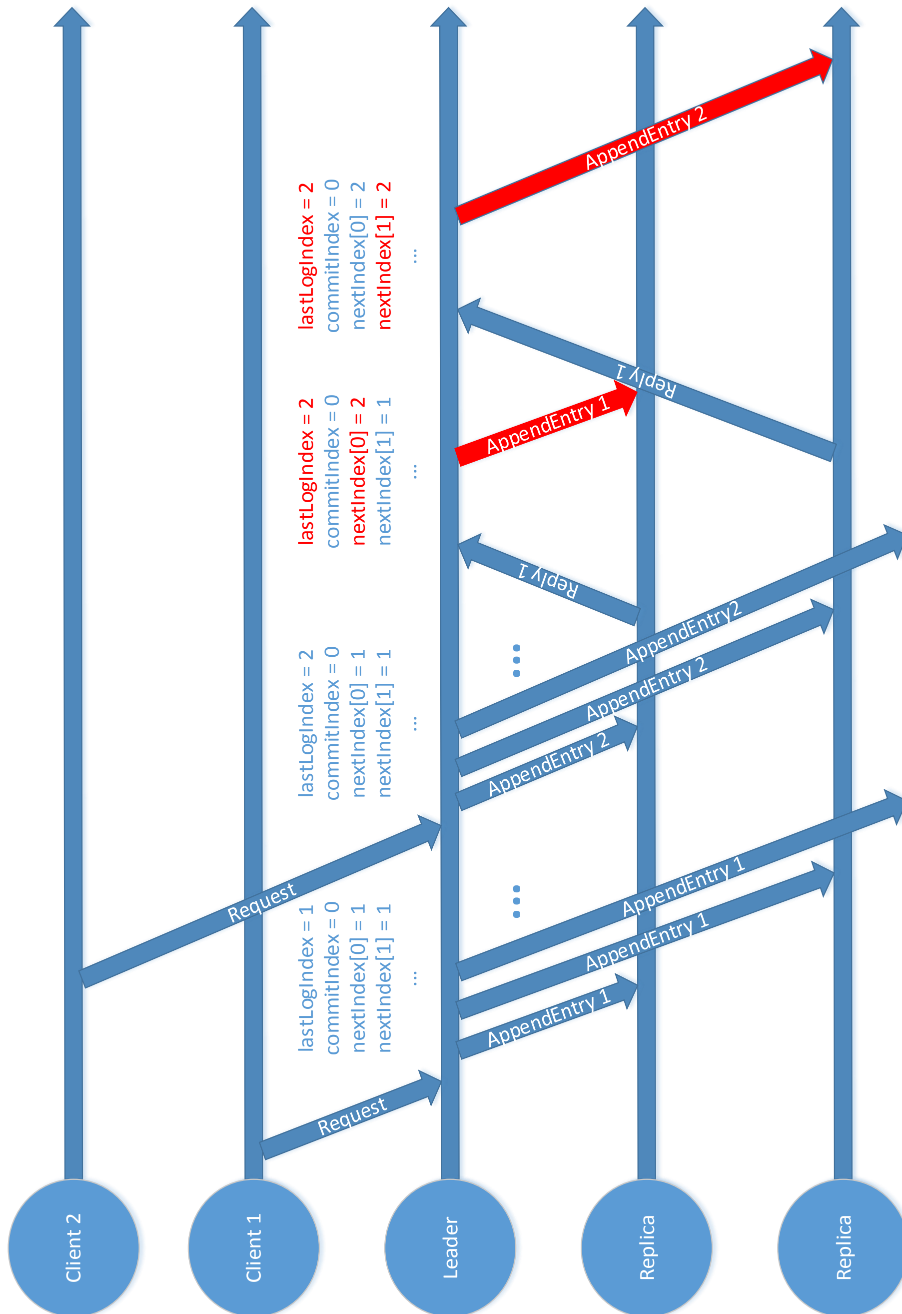
The implementation of Raft required more design decisions than 2PC. Raft requires two RPCs: *AppendEntries* and *RequestVote*. To avoid blocking, we split up the RPCs into two messages (request, reply). Handling blocking RPCs requires that we start the RPC in a thread. Otherwise if a replica fails, an RPC to this replica blocks the whole agreement. Even more, detecting and aborting an RPC that timed out requires further mechanisms. We concluded that using RPCs may not give us the performance we want when running on a multicore environment. Additional to the asynchronous messages, we had to limit the number of entries we send per message to one to avoid messages larger than a cache line. If Raft is used over a network, all these problems do not appear since computation time is not dominating the cost.

When messages are not handled as RPCs but as two asynchronous messages, it is enough to detect the failure of a replica (part of the Raft protocol). We tried the implementation with two asynchronous messages per RPC but some other problems occurred. Two asynchronous messages weaken the strong coherency between leader and followers.

First, a small recap of the meaning of different values. The leader updates the `commitIndex` when a majority of replicas contains the entry in their log. The replicas update their `commitIndex` when they receive messages from the leader. The `lastLogIndex` for each replica is the last entry that this replica has appended to the log. The leader keeps track of the next log index to send to a follower with the `nextIndex[]` array. The leader resends entries to a replica if `lastLogIndex` $\geq$ `nextIndex[]` holds for a follower.

Let us assume that there are two clients and each of them sends a request to the leader. The leader receives both requests and broadcasts an *AppendEntry* message to all the replicas. The leader's `lastLogIndex` starts at 0 and after the two requests it is equal to 2. The `commitIndex` is still 0 since the leader has to receive at least a majority of replies to increase the `commitIndex`. The leader can not increase the `commitIndex` after receiving the first response to the *AppendEntry* message. The leader updates its `nextIndex` for this follower to 2.

Figure 6.2: Raft: problem of additional messages



Since `lastLogIndex` $\geq$ `nextIndex`[1] the leader sends an *AppendEntry* message for the entry 2. The addition message is redundant since the leader has already sent an *AppendEntry* message for this entry.

In figure 6.2, the red messages are not necessary. The problem gets worse with more clients since the gap between `lastLogIndex` and `nextIndex[]` can grow larger (up to the number of clients). We did not come up with a solution for this problem. It does not happen if normal RPCs are used but as explained before, RPCs in this context present another set of problems to solve efficiently. We could not solve the problem in a timely and efficient fashion and did not include Raft as a protocol into our framework.

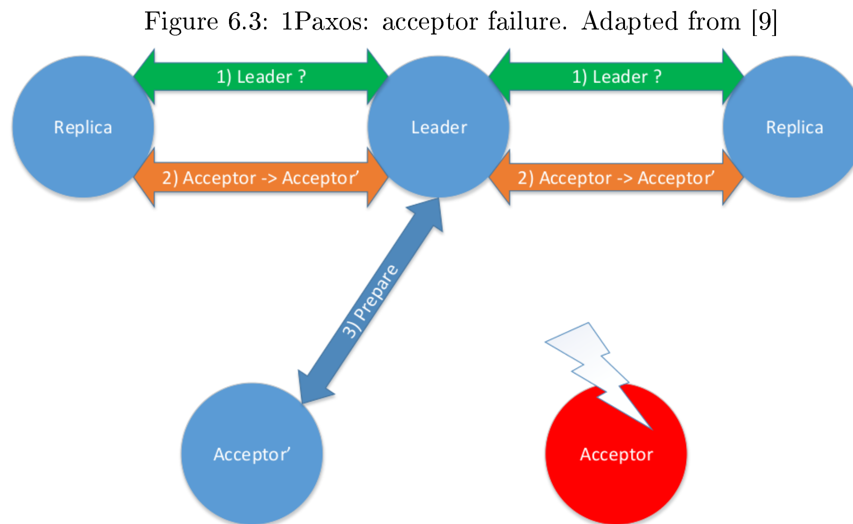
6.3 1Paxos

In contrast to Raft, 1Paxos does not have a lot of implementation details in the paper [9]. The base case for 1Paxos is straight forward but dealing with failures is harder. The failure of either acceptor or leader involves detecting the failure as well as finding a replacement.

6.3.1 Acceptor Failure

Before starting the procedure to change the acceptor, a replica has to detect that the acceptor failed. The idea is similar to Raft. The leader detects the acceptor failure by starting a periodic event that sends a message to the acceptor. If the acceptor does not respond in a timely fashion, the leader assumes that the acceptor failed. If an acceptor fails, the protocol takes the steps shown in figure 6.3.

Figure 6.3: 1Paxos: acceptor failure. Adapted from [9]

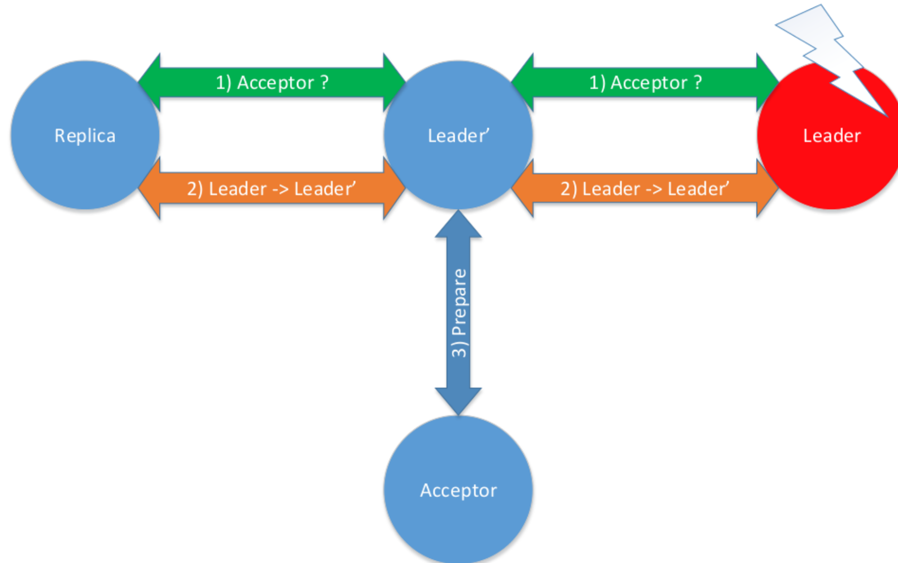


In the paper they suggest running an instance of Multi-Paxos to decide on an acceptor. Implementing another failure tolerant protocol to just select a new acceptor was too much for us. We chose a simpler approach and let the leader decide on the new acceptor. After finding a new acceptor, the three steps shown in figure 6.3 are executed. To avoid having two leaders that change the acceptor, a leader has to request the id of the current leader from all replicas. If the leader receives a majority of messages with its id, the leader can proceed and announce the change.

6.3.2 Leader Failure

Leader failure has the same issues as acceptor failure: detecting the failure and choosing a new leader. We solved the problem with a similar mechanism as used in Raft. The failure of the leader is detected by periodic messages that are sent to the leader from all the replicas except the acceptor. If a replica does not receive an answer in a timely fashion, the replica assumes that the leader failed and the replica tries to become leader. To avoid two replicas trying to become leader at the same time, we added a random backoff time. The steps taken after failure detection and finding a new candidate for the leader are shown in figure 6.4.

Figure 6.4: 1Paxos: leader failure. Adapted from [9]



In the first step the replica trying to become leader contacts all other replicas to receive the current acceptor id. After a replica sends the acceptor id to the new leader, it does not respond to other replicas trying to inquire the acceptor id to avoid having two leaders. In the case of a draw, the replicas reset their vote

and after some time, the process begins again. After the three steps shown in figure 6.4, the new leader informs the clients to send their requests to it instead of the old leader.

6.3.3 Flooding Message Channels

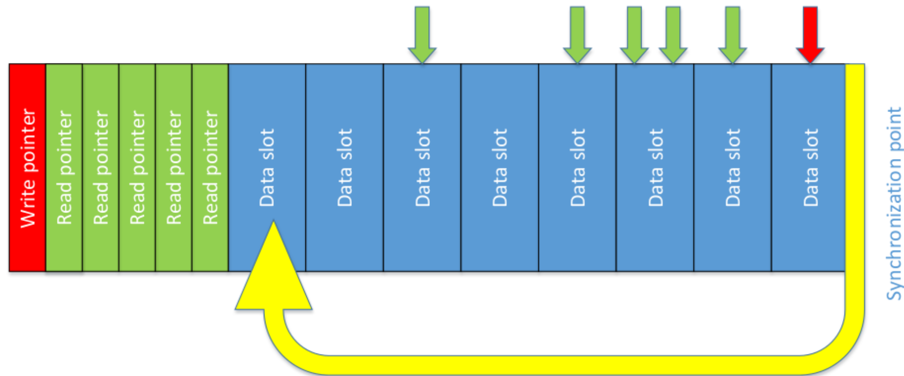
1Paxos does not require any acknowledgements from the replicas to the acceptor. The leader replies to the client as soon as the leader receives a *learn* message. Since the broadcast of the *learn* message from the acceptor to the replicas is sequential, the leader is often one of the first replicas to receive it. As explained before, in Barrelfish a communication channel only has a buffer for a single message. In the case of 1Paxos, it can happen that the channel is full. We handle a full channel by allocating some memory and caching the messages in a queue. If the channel is free, the cached messages are sent one after another. Since there is no acknowledgement back from the replicas, the acceptor may send messages at a faster rate than the other replicas are able to handle. In this scenario, the queue can grow to arbitrary length.

We tried to inform the leader with a *learn* message as the last replica, but the problem still occurred. Even if the message to the leader is sent last, it is not given that it is the last one to finish processing the message. Since we are not expecting that our implementation is always running at its maximal throughput, we send the message to the leader as one of the first messages. The benefits of sending the message early to the leader outweigh under normal execution.

6.4 Single-writer Multiple-reader Queue

To get a general solution for the shared memory part of the framework, we implemented a Single-writer Mutliple-reader (SWMR) queue. The queue consists of two parts: meta data and data slots. The meta data contains the positions for the writer and readers within the circular queue of data slots. The layout within memory is shown in figure 6.5.

Figure 6.5: Shared memory queue: layout in memory



At the end of the queue, there is a synchronization point where the writer does not proceed until all the readers caught up. The synchronization point prevents the writer from overwriting data that is not processed by the readers yet. The writer and reader pointers are all of the size of a cache line to avoid false sharing.

To replicate data, the writer is started on the replica that participates on the higher level protocol. The readers are started on all other cores within a NUMA node and are polling for new data. The interface to use the queue is shown in listing 6.1.

Listing 6.1: Shared memory: single-writer multiple-reader queue interface

```
void shm_write(void* addr);
void* shm_read(void);
```

The function `shm_write()` is only called by the writer and does not require a lock. The reader polls the queue with `shm_read()`. If the reader calls `shm_read()` and the reader is already at the position of the writer, `shm_read()` returns `NULL`. If the returned slot is unequal to `NULL`, the reader executes the function similar to a normal replica.

We generalized our SWMR queue so it can be used between nodes. The writer of the queue implements the uniform client interface. Any request from any core is sent to the writer by message passing. The writer and readers each have their own cache line that only they modify. Only the writer modifies more than one cache line by writing into data slots. The only problem when replicating data should be the queueing of messages at the writer.

6.5 Broadcast Replication

Within a node not only shared memory is an option but message passing as well. We implemented a simple broadcast from a leader to all replicas within a node. Similar to the shared memory queue, the broadcast replication was intended to be used within a node and was later generalized that it can be used between nodes.

To avoid putting more load onto the replica that already participates in the higher level protocol, the replica on the second core within a NUMA node has a special role. In the following we call this replica the leader. The leader receives the requests handed over from the replica of the higher level and broadcasts them. It replies to the replica after it finished broadcasting the request. Under high load, the leader suffers from the same issue as our 1Paxos implementation. The leader may try to broadcast before the replicas received and processed the previous messages.

The broadcast replication can be started on the higher layer. It can be combined with any of the other protocols and algorithms running on the lower layer. Since the broadcast replication does not handle failures, we implemented both a sequential as well as a tree broadcast.

6.6 Composability

Composing different protocols presented us with two problems. One is the setup of all the replicas on the right cores. We do not explain this in detail since most of it is parsing arguments and spawning the replicas in a new domain. The bigger problem is forwarding the agreed values from the higher to the lower layer.

After the higher layer has agreed to a certain value, the lower layer protocol is started until we reach the bottom of the hierarchy. The interface between the layers is blocking. When the higher layer forwards the agreed value to a protocol instance on the lower layer, it waits until the replication on the lower layer is finished. To avoid putting more load on the higher level replicas, they do not participate as a replica in the lower level protocol. The higher level replicas simply send requests to the lower layer replicas as a client.

We tried different implementations for handling the communication to the lower layer. From the higher layer to a cluster of the lower layer there is only a single communication channel (or client). With more than one channel, there is the possibility that messages may overtake each other. If a message overtakes, it invalidates the order of agreements of the higher layer and correctness is not guaranteed. If there is only a single channel, it can be overrun by too many messages since the higher layer receives requests from several channels. The requests that can not yet be served by the channel have to be queued. We avoided

arbitrary growing queues by blocking until the lower layer is finished with the replication.

When a lower layer protocol is started, we initialize something similar to a client. A client is necessary to forward the request from the higher layer to the lower layer. The code at the moment does not make use of the uniform client interface as shown in chapter 5. We implemented the layer for composability before coming up with the idea of a uniform client interface. The time it takes to change the implementation is not the problem, but validating that all combinations of protocols work correctly is time consuming. Out of this reason we postponed the change since it does not add any functionality.

6.7 Key Value Store

We implemented a key value store to showcase our framework. The keys and values are both 64 bit integers. The key value store replica can be implemented in as few lines of code as shown in listing 6.2.

Listing 6.2: Key value store: simple implementation

```
static void exec_function(void* addr)
{
    struct consensus_payload* cmd;
    cmd = (struct consensus_payload *) addr;
    kv_start[cmd->arg1] = cmd->arg2;
}

static int client_loop(void* arg)
{
    // wait until replicas are setup
    barrelfish_usleep(10*1000*1000);

    // requets loop
    while (true) {
            // put client logic here that puts arguments into
            // consensus_payload struct and sends the request
            // and reads values
            ...
    }
    return 0;
}

int main(int argc, char ** argv)
{
    // allocate key value store memory
    void* buf = ...
    ...
    ...

    kv_start = (uint64_t* ) buf;

    // init replica
```

```
    init_consensus_replica_argv(argv, exec_function);

    // init client logic (optional)
    init_consensus_client();
    thread_create(client_loop, NULL);

    // start handle messages
    start_handler_loop();
}
```

The whole key value store with a replica and a client on each core can be started by simply calling `consensus_init_auto()` with the string of the key value store's program. The main function of the program just allocates some memory for the key value store and initializes the replica in this domain. After the replica is set up, a client to send requests to the replicas is initialized and the client logic is started.

The simple implementation shown in listing 6.2, is to showcase the usage of the framework and does not have an interface. The interface to a more complex key value store is shown in listing 6.3.

Listing 6.3: Key value store: interface for clients

```
errval_t init_kv_store_client(void);
errval_t kv_store_write(uint64_t key, uint64_t value);
uint64_t kv_store_read(uint64_t key);
```

The replicas started for the key value store implement a Flounder interface that lets clients request the capability of the key value store's memory. The function `init_kv_store _client()` just requests the capability from the first core of the NUMA node and maps it into its address space. Additionally, the key value store client initializes a client for the agreement service. The function `kv_store_write()` marshals the key and value into a `consensus_payload` struct and sends the request to the service while the `kv_store_read()` function simply reads from the locally mapped memory.

The capability of the key value store's memory is requested from the first core of a NUMA node. In this manner, the key value store is still functional even if we decide to reduce the number of replicas to one per NUMA node. The key value store can be directly replicated within a node by mapping the capability into the address space.

Chapter 7

Evaluation

In this chapter we evaluate how the improvements we found and the composition of protocols affect performance. Finally, we assess our framework with the implementation of the key value store.

7.1 Setup

In the evaluation chapter we use several machines. The benchmarks on Linux are run on Magny-cours while the Barrelfish benchmarks are run on Istanbul and Ivy-bridge. The specifications of these machines are shown in table 7.1.

Table 7.1: Evaluation: benchmark machines

Magny-cours	AMD Opteron 6174 4×12 cores 2.2GHz, 128 GB RAM
Istanbul	AMD Opteron 8431 4×6 cores 2.4GHz, 16 GB RAM
Ivy Bridge	Intel Xeon E5-2670 v2 2×10 2.5GHz, 256 GB RAM

The NUMA node assignment of all the machines is shown below.

Table 7.2: Evaluation: cores assigned to NUMA nodes

Node Number	Magny-cours	Istanbul	Ivy Bridge
Node 0	0,4,8,12,16,20	0-5	0-9
Node 1	24,28,32,36,40,44	6-11	10-19
Node 2	1,5,9,13,17,21	12-17	-
Node 3	25,29,33,37,41,45	18-23	-
Node 4	2,6,10,14,18,22	-	-
Node 5	26,30,34,38,42,46	-	-
Node 6	3,7,11,15,19,23	-	-
Node 7	27,31,35,39,43,47	-	-

For all the experiments (except stated otherwise) the clients do not sleep between requests. Each benchmark consist of a 20 second warm-up period and three measurements over 20 seconds.

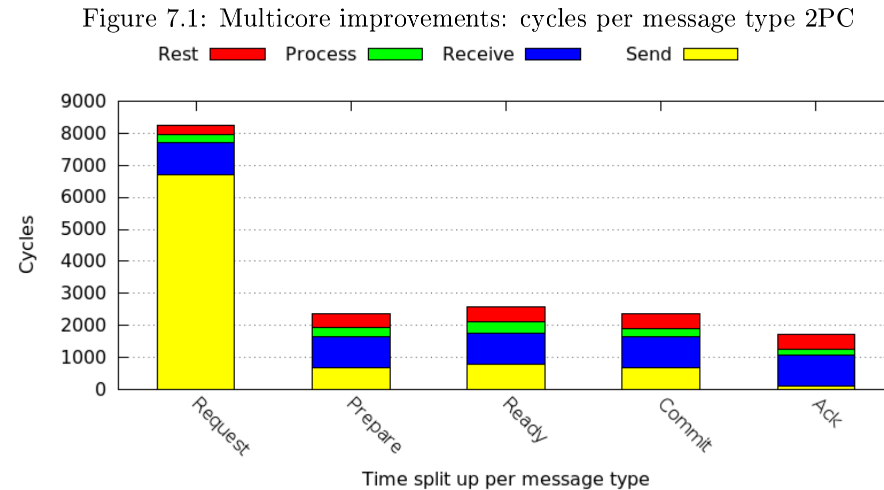
7.2 Improvements for Multicore

In a first comparison we take the values measured with the QC-libtask framework and their 2PC implementation and compare it to our 2PC implementation. Our implementation uses a tree broadcast and reduces the message processing overhead. In a second part, we change the underlying operating system.

7.2.1 Tree Broadcast and Message Processing

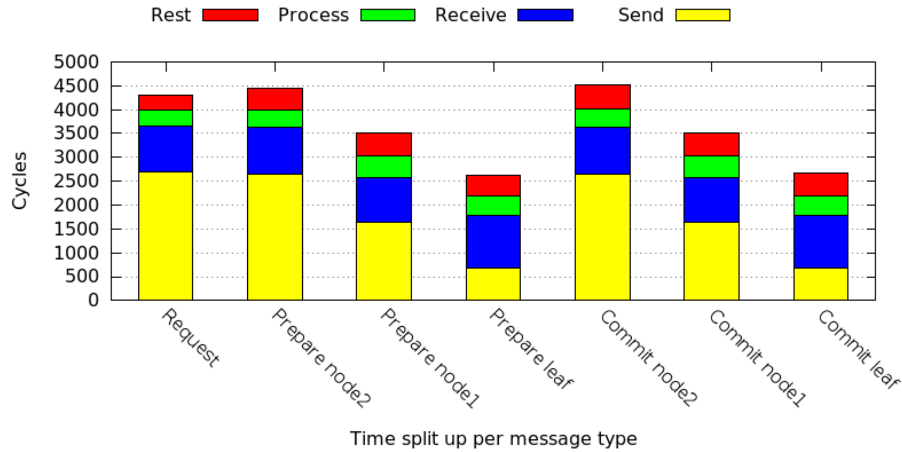
The benchmark is run on a Magny-course machine. We started 8 replicas and a single client for the micro benchmarks. Figure 7.1 shows the time spent in different code segments of our 2PC with a sequential broadcast.

Figure 7.1: Multicore improvements: cycles per message type 2PC



In our 2PC implementation with a sequential broadcast, the number of cycles for processing a message reduces from around 1'500-2'000 to a mere 300-400. The time it takes to receive and send messages dominates (excluding wait time). The time spent in code segments for a tree broadcast is shown in figure 7.2.

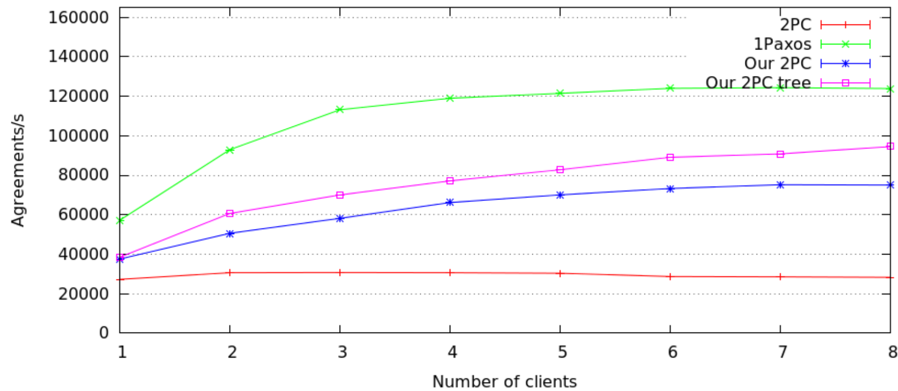
Figure 7.2: Multicore improvements: cycles per message type 2PC using broadcast tree



The load from sending messages is more evenly distributed between replicas. The number of messages a replica forwards changes depending on the position within the tree. The leader requires around 4'000 cycles less to broadcast a message.

In figure 7.3 we compare the throughput of different implementations of 2PC with 1Paxos.

Figure 7.3: Multicore improvements: throughput of implementations based on QC-libtask



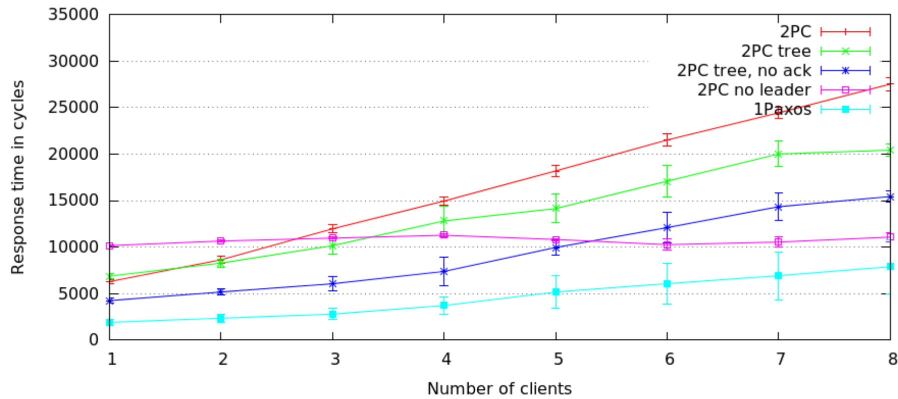
The maximal throughput of 2PC already increases by almost a factor of three by removing the overhead from processing the messages. Another 20'000 agreements per second are added to the maximal throughput by broadcasting along

a tree. Still, the throughput of 1Paxos is unreachable with our 2PC. The 2PC protocol, in comparison to 1Paxos, requires too many messages to reach an agreement.

7.2.2 Barrelfish Implementations

There is only one problem left to tackle: reducing the time needed for selecting the right thread to run. We changed the operating system from Linux to Barrelfish. Barrelfish offers a lot of features concerning message passing and handling messages. The experiments are performed on an Ivy Bridge machine using 8 replicas and a varying number of clients. The response time of our 2PC implementation based on QC-libtask varies from 65'000 cycles for 1 client to 268'000 for 8 clients. We did not include these numbers for readability reasons. The other results are shown in figure 7.4.

Figure 7.4: Barrelfish: response time of different protocols w.r.t. number of clients

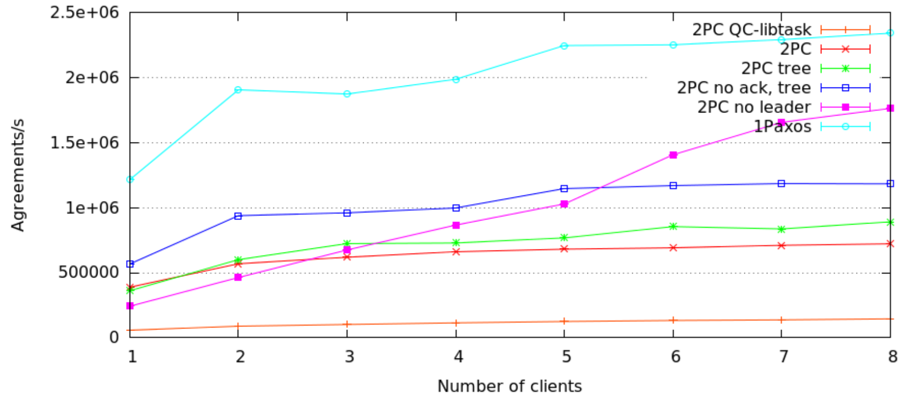


The response time of the protocols reduce to reasonable numbers. Most of the difference is accountable to the change to waitsets and their event handling mechanisms. The time it takes from sending a message to receiving and processing the message on the other core reduces. Additionally, the message passing of Barrelfish (UMP) is faster than the QC-libtask framework.

We benchmarked four different 2PC implementations. The basic 2PC with no improvements, a 2PC using a tree broadcast, 2PC that does not acknowledge the last message, and a 2PC without a leader. The benchmarks do not consider conflicts for 2PC without a leader. If there are conflicts, the performance of 2PC without a leader reduces. The adjustments of 2PC to a multicore machine reduce the response time. Reducing the number of messages by not sending an *ack* has the largest impact on the performance of 2PC. The higher the load, the larger the gap between the improved 2PC to baseline 2PC. The 2PC that

does not rely on a leader has a stable response time since the clients execute the broadcast. The replicas in the best case only receive two messages and send one message. The response time suggests that the replicas are not fully loaded yet and there are resources left to handle more clients. 2PC without a leader is an option if the throughput of the system is of the essence. The lowest response time is measured from 1Paxos and shows what adapting a protocol to a multicore environment can deliver. The throughput shows the same picture.

Figure 7.5: Barrelfish: throughput of different protocols w.r.t. number of clients



In figure 7.5 we included the numbers for our 2PC implementation based on QC-libtask for comparison. For a higher number of clients 2PC without a leader can rival 1Paxos. The broadcast of 1Paxos is still executed on the acceptor in contrast to 2PC without a leader. The broadcast on the clients is an advantage if a high throughput is important.

7.3 Failures

1Paxos is able to recover from non Byzantine failures as long as a majority of replicas is running. The benchmarks in this section are performed on an Ivy Bridge machine (2×10 cores). In the first experiment (figure 7.6) we monitored the response time of a single client during a leader failure. The values were measured after a warm-up of 1'000'000 requests.

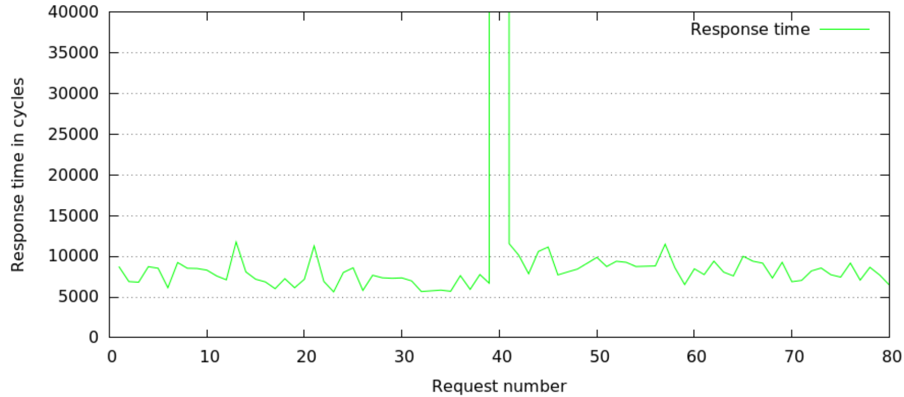


Figure 7.6: Failures: response time during recovery from a failure

The response time does not change dramatically after the new leader is chosen. There is a slight increase after the failure but the response time slowly reduces back to the normal average.

In a next benchmark we show the performance of 1Paxos depending on how many failures it can tolerate. We increased the number of replicas from 3 (1 failure) to 15 (7 failures). We benchmarked with both the load of 4 clients and 8 clients.

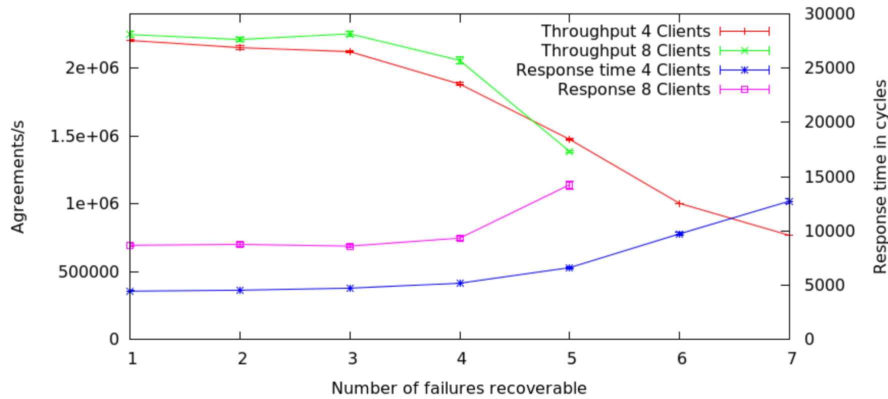


Figure 7.7: Failures: performance w.r.t. number of failures recoverable

The throughput and response time do not change until 1Paxos can recover from four failures. A different protocol than 1Paxos would have more problem to scale. As a reminder: 1Paxos responds back to the client as soon as the leader receives a *learn* message. The quick reply reduces the response time and 1Paxos can scale to a higher number of replicas. As soon as 1Paxos can recover from five failures and a replica is running on the 11th core, the acceptor becomes a

bottleneck. The requests from the clients are queued at the acceptor while it is busy sending messages to the replicas.

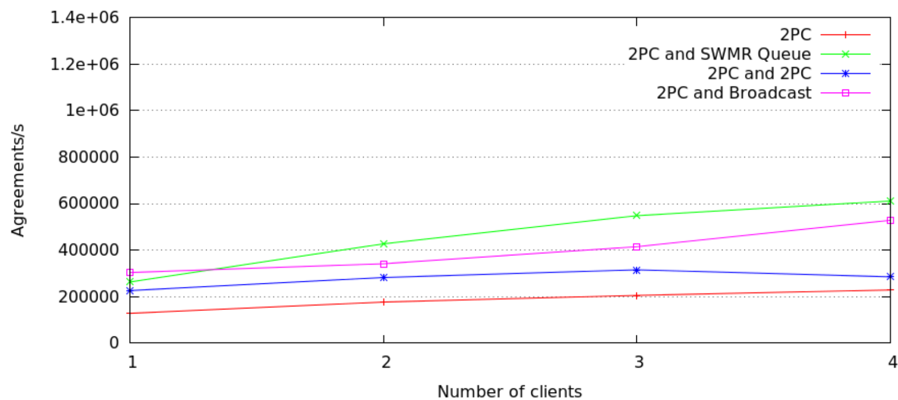
7.4 Composition

Scaling to a higher number of replicas is a problem that can be solved by a composition of protocols/algorithms. All the benchmarks of this section are run on an Istanbul machine (4×6 cores). The replicas that communicate between the nodes, are pinned to the first cores of the NUMA nodes (0,6,12,18). The rest of the cores start another protocol/algorithm except for the last core of each node (5,11,17,23). The last core of each node is putting load onto the system by starting a client. Overall, the setup adds up to 20 replicas and up to 4 clients. The graphs in this section have the same scale and a direct comparison is possible.

7.4.1 2PC

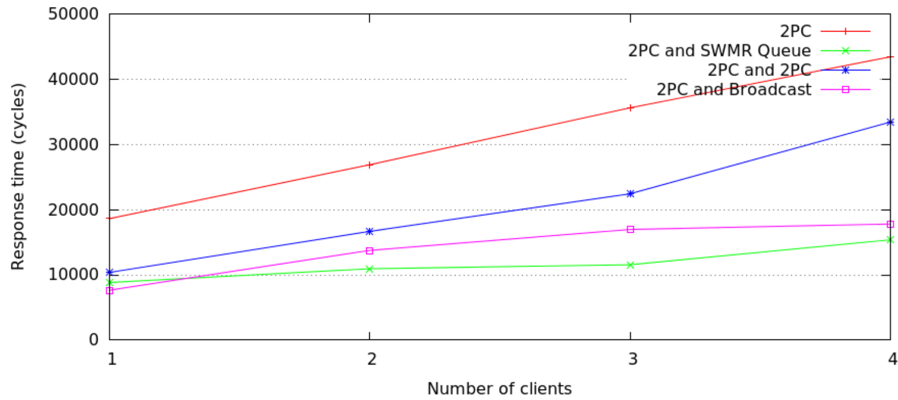
The base line is 2PC on all 20 cores. The rest of the benchmarks combine 2PC with either shared memory (SWMR queue), 2PC itself or the broadcast replication. The throughput is shown in figure 7.8.

Figure 7.8: Composition: throughput of compositions of 2PC w.r.t. number of clients



Any composition that splits up the problem yields better results than solving the whole problem at once. 2PC on both layers has a higher throughput than 2PC over all 20 cores. At a lower load, broadcast replication has a higher throughput than the combination with a SWMR queue. Broadcast replication within a node is a good alternative to shared memory. The response time in figure 7.9 shows similar results.

Figure 7.9: Composition: response time of compositions of 2PC w.r.t. number of clients

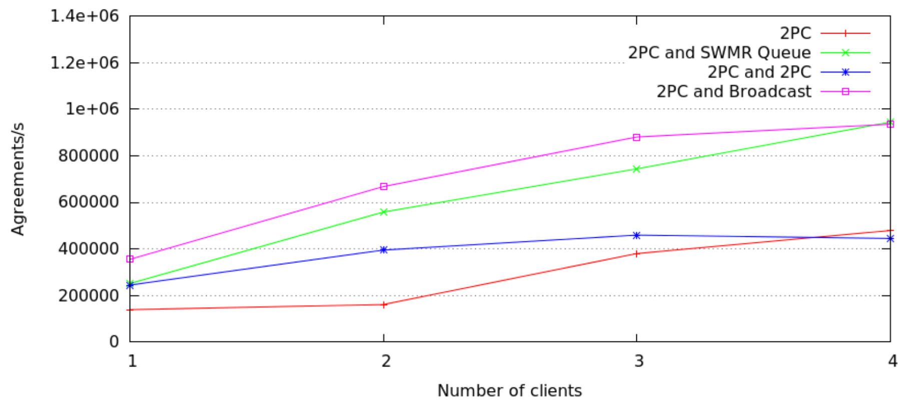


The slope of the response time of 2PC on all cores is the largest. Here we can see the problem with 2PC and its leader. Most of the load is on the leader regardless of using a tree broadcast. The leader has to process most of the messages and becomes a bottleneck. The problem becomes more clear when looking at 2PC without a leader.

7.4.2 2PC without Leader

The compositions are the same as before. The throughput of 2PC without a leader and its combinations are shown in figure 7.10.

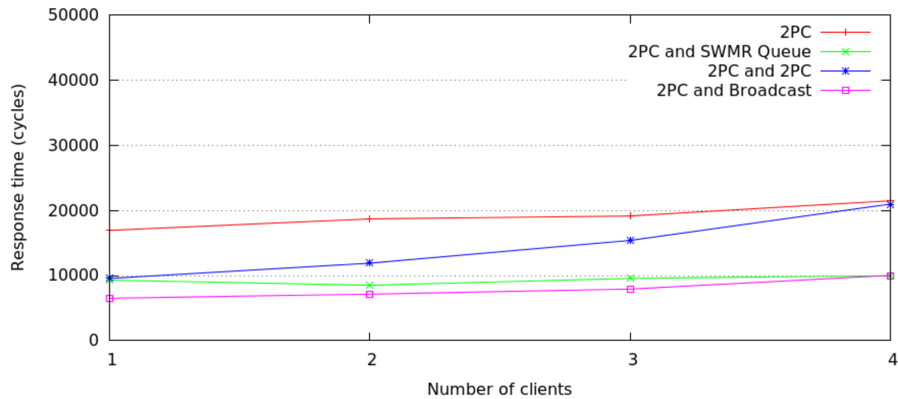
Figure 7.10: Composition: throughput of compositions of 2PC without leader w.r.t. number of clients



2PC without a leader puts the load of the broadcast on the client. In the benchmarks we only started 4 clients which is not enough load for 2PC to reach

its limit. Running 2PC on all 20 cores performs better as the number of clients increases since the client performance is the limiting factor. 2PC on both levels has the problem that from the higher level to the lower level the broadcast is executed on the core of the replica of the higher level. The highest throughput is reached by the combinations with either the SWMR queue or the broadcast replication. The response time of the combinations is shown in figure 7.11.

Figure 7.11: Composition: response time of compositions of 2PC without leader w.r.t. number of clients

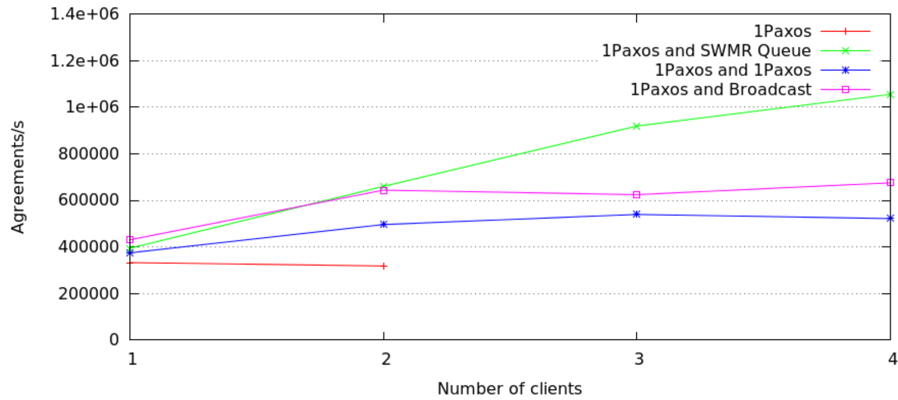


The response time of all combinations does not get notably worse if the number of clients increases. The only exception is 2PC without leader on both levels of the hierarchy because of the increased load on the higher level replicas. The combination with the broadcast replication has the lowest response time. When the SWMR queue is not used on the lower layer, sending of requests to the lower layer does not completely block. Other messages are processed as long as there is no reply from the lower layer. The first round of messages of 2PC are processed while waiting for the response of the lower layer. We assume that this is the reason why broadcast replication has a lower response time than the SWMR queue.

7.4.3 1Paxos

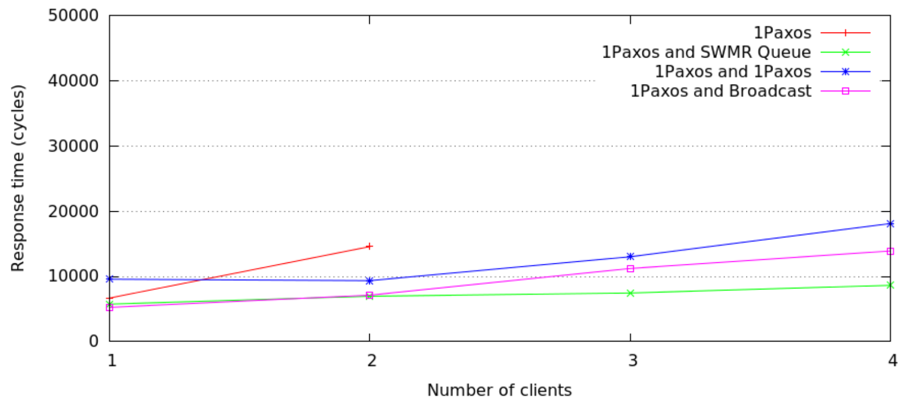
As explained in subsection 6.3.3, 1Paxos suffers from a problem at high load (gets worse with more replicas). The problem prevents us from showing reasonable results for more than two clients. The problem does not occur for compositions since the lower level blocks for some time, letting enough time for the replicas to process the messages. The throughput results are shown in figure 7.12.

Figure 7.12: Composition: throughput of compositions of 1Paxos w.r.t. number of clients



The sequential broadcast of 1Paxos on all 20 cores does not scale. The combination with the broadcast replication is again in second place. Still, the performance difference to the SWMR queue is larger than with any other protocol. The most scalable combination is 1Paxos and the SWMR queue reaching over a million agreements per second. All the protocols show that shared memory on the lower layer is a good choice but for some protocols the broadcast replication is an alternative. The response time is shown in (figure 7.13).

Figure 7.13: Composition: response time of compositions of 1Paxos w.r.t. number of clients



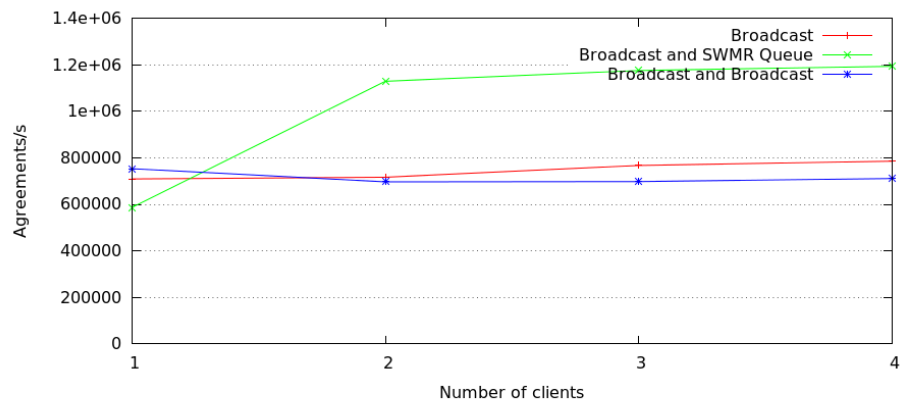
1Paxos on all 20 cores shows a similar slope as the normal 2PC when increasing the number of clients. The response time of the compositions not using shared memory increases after the system is saturated. The only combination that can stay below 10'000 cycles for 4 clients is 1Paxos with the SWMR queue. The small increase in response time from 1 to 4 clients suggests that the system is

not yet at its limit.

7.4.4 Broadcast Replication

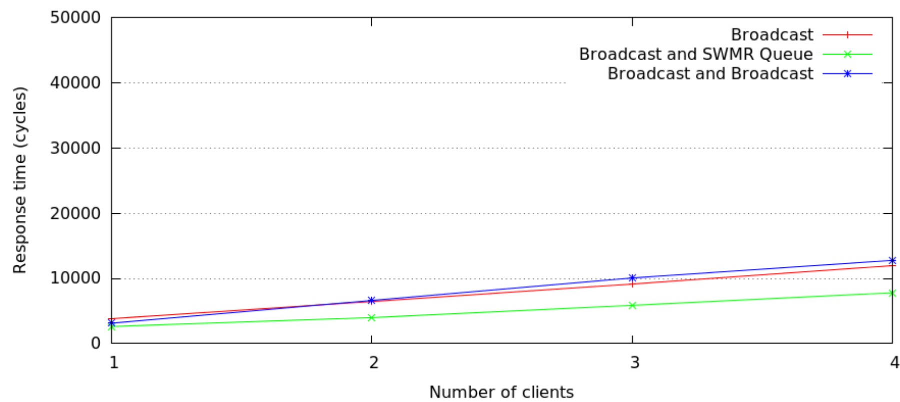
Broadcast replication is the simplest way of replicating by message passing. Consequently, it is the fastest of our implementations. The throughput of the compositions of broadcast replication is shown in figure 7.14.

Figure 7.14: Composition: throughput of compositions of broadcast replication w.r.t. number of clients



Only the throughput of the combination of broadcast replication and the SWMR queue increases if there is more than one client. The throughput is higher than 1Paxos combined with shared memory. The difference between broadcast replication on all 20 cores and the two level broadcast replication is small. The response time is shown in figure 7.15.

Figure 7.15: Composition: response time of compositions of broadcast replication w.r.t. number of clients



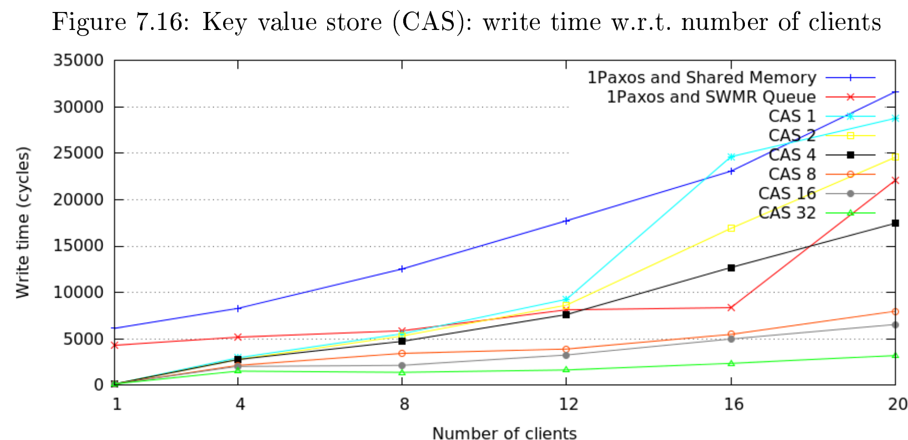
The response time of broadcast replication combined with the SWMR queue is around 2'600 cycles for one client. The lowest response time we measured from the compositions with 1Paxos is around 5'000 cycles. Broadcast replication has the best performance and strengthens our choice if no failure tolerance is needed. The response time of the compositions, when increasing the number of clients, only differs in the slope.

7.5 Key Value Store

As a last part of our evaluation we show the performance of our key value store. We used an Istanbul machine to run the benchmarks. We compare two version of our key value store to a key value store based on shared memory. In one of the versions, the key value store is replicated on all cores by a combination of 1Paxos and the SWMR queue. In the second setup, we only start 1Paxos between nodes and directly share the memory of the key value store within a NUMA node.

We compare the two setups to a key value store based on only shared memory and Compare-and-swap (CAS) for updates. An implementation based on CAS is possible since there is only a single 64 bit value that changes per update. The performance of CAS depends on the number of different memory locations that are written. The less different memory locations, the higher the contention with CAS. Each key represents a memory location in the key value store. We benchmarked with a number of keys ranging from 1 to 32. The benchmark randomizes the writes to the keys. We do not compare the read time since there is almost no difference between the implementations. The clients are running in the same domain as the replicas but we do not start clients on the cores of the higher level replicas. The write time is shown in figure 7.16.

Figure 7.16: Key value store (CAS): write time w.r.t. number of clients

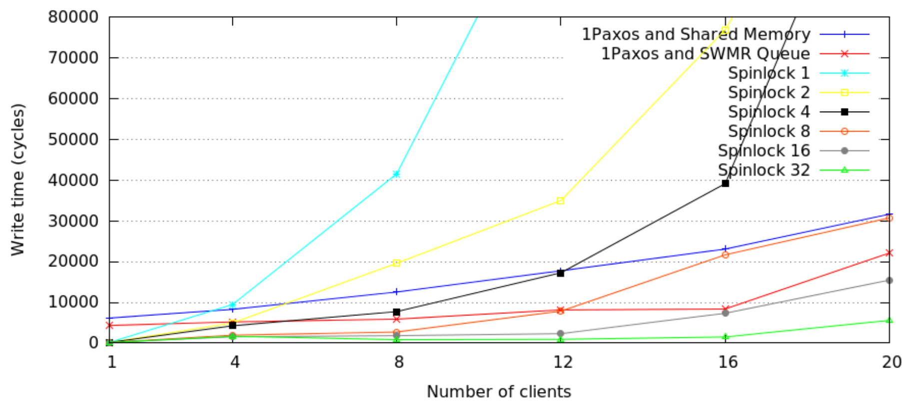


The response time of the CAS based key value store increases as the number

of different memory locations reduces. The response time of 1Paxos combined with the SWMR queue is lower than the response time of 1Paxos (directly share the memory). 1Paxos combined with the SWMR queue has the problem of scheduling the replicas besides the clients. Consequently, most of the time is spent scheduling between requests resulting in a low load on the system. 1Paxos between nodes and directly sharing the memory within nodes is running at a much higher load that leads to queueing of requests and an increase in response time. Nevertheless, CAS has the lowest response time. CAS can make use of the parallelism of a multicore machine, while any consensus protocol is limited in this regard. Overall, a consensus protocol provides more guarantees (sequentializability, fault tolerance) that are not strictly needed in a key value store.

CAS can only exchange a single value of 8 byte at a time. Our implementation based on consensus protocols is less restricted and can write values of 16 bytes at a time (if required). We implemented a second version of a key value store based on shared memory that relies on spinlocks rather than CAS. Spinlocks are necessary since the exchange of two values has to happen atomically. The second version allocates one spinlock per 10 keys. A lock per 10 keys is a usual trade-off for key value stores. Similar to CAS, the performance is dependant on the number of different memory locations. Since a spinlock is simply a memory location, the performance is dependant on the number of spinlocks since they are accessed more frequently. We benchmarked in the range of 1 to 32 spinlocks (10 to 320 keys) and randomized the writes. The write time is shown in figure 7.17.

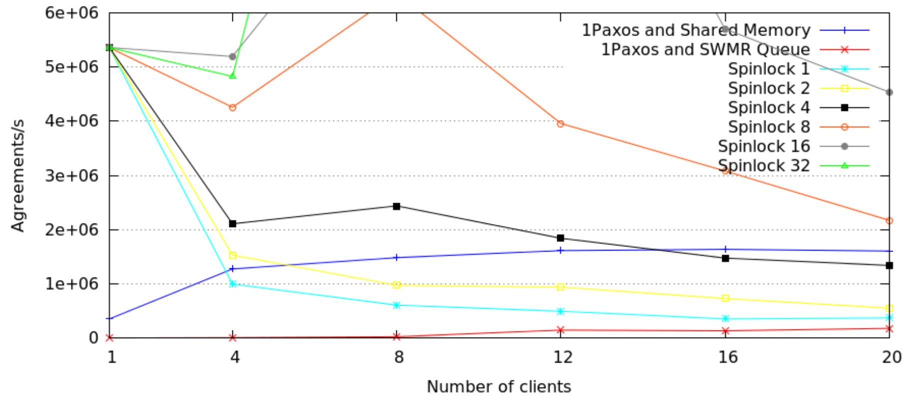
Figure 7.17: Key value store (spinlock): write time w.r.t. number of clients



If the number of spinlocks decreases (or number of memory locations), the response time increases. As expected, the response time of the key value store with spinlocks is higher than the previous implementation based on CAS. The response time of the implementations based on our framework are more competitive. For a high number of clients combined with a low number of updated

keys, we can compete with spinlocks. For a low number of spinlocks, the clients more often access the same spinlock and the response time can get arbitrary bad. The throughput reflects these findings (figure 7.18).

Figure 7.18: Key value store (spinlock): throughput w.r.t. number of clients



The highest throughput of 1Paxos combined with the SWMR queue (20 clients) is only around 150'000 agreements per second because of the scheduling problem. By directly sharing the memory within a node and avoiding the scheduling problem, the throughput with 20 clients is around 1.6 million. The spinlock based implementation outperforms our framework for a higher number of spinlocks. If the contention of the spinlocks increases, either because of fewer memory locations or higher number of clients, the throughput reduces. The highest throughput we measured is around 22 million for 32 spinlocks (cut off for readability reasons).

Shared memory can make use of an increased parallelism leading to a higher throughput. There are cases when shared memory reaches its limit. Because of synchronization mechanisms the performance under high contention can get arbitrary bad. Our framework performs reasonable when many clients update few memory locations.

Chapter 8

Related Work

We have already presented the relevant consensus protocols in chapter 3. Other papers concerning our work are summarised in this chapter.

8.1 Rex

Rex [12] is another approach at optimizing replication to a multicore environment. The Rex framework introduces a new consistency model called *execute-agree-follow* to reach a higher degree of parallelism. In a first step, Rex uses a primary replica that executes requests in parallel and produces a partially ordered trace of requests. If the primary fails, another replica is promoted to the primary replica. In a second step, Paxos is run between the replicas to agree on traces. Paxos prevents inconsistencies if it happens that there are two primaries. In a last step, the replicas execute the trace they agreed on. After the execution of the trace, the replicas are consistent with the primary.

8.2 Composing Shared-Memory Algorithms

A single shared-memory algorithm can never guarantee efficiency under all conditions. Composing different shared-memory algorithms to preserve the best performance under different circumstances is a solution to this problem. In the paper "On the Cost of Composing Shared-Memory Algorithms" [1], Dan Alistarh et al. show that different shared-memory algorithms can be composed, if these algorithms are implemented as *safely composable*. If an algorithm is no longer optimal or stalls, the algorithm can be aborted and forward some of its state to the next *safely composable* algorithm. The state transferred contains a sequence of requests (history) that was previously executed. The more optimal algorithm is initialized using this history.

8.3 Optimizing Collective Communication on Multicores

Rather than optimizing communication patterns by hand to a certain platform, automatic tuning [21] leads to algorithms that are scalable and deliver good performance. In the paper of Nishtala et al. [21] mainly the synchronization patterns *Barrier* and *Reduce* are discussed. A call to a *Barrier* does not exit until all the threads reach the *Barrier*. The communication topology of a *Barrier* can be implemented as a tree, improving the performance. Another collective synchronization effect can be studied by looking at the *Reduce* operation. A *Reduce* operation lets threads combine their results to a global result. Two different synchronization modes are considered: loose and strict. In a loose synchronization mode from the first thread entering the collective data movement can begin. When taking a strict approach, all threads have to enter the collective to start the data movement. By combining the collective tuning and loose synchronization the best performance is achieved.

Chapter 9

Conclusion

In a first part, we had a look at what changes from a network environment to a multicore environment. We defined a failure model and failure domain for our work. Based on these assumptions, we evaluated different consensus protocols and chose 2PC, 1Paxos and Raft. Moreover, we investigated how consensus protocols can be improved to better match into a multicore environment by analysing the framework implemented by Rachid Guerraoui et al. [9]. We found three problems: the cost of sequential broadcasts, the cost of processing messages, and the overhead for selecting a thread to run.

In a second part, we presented our consensus framework design. With the help of the framework we were able to combine different consensus protocols. Combining protocols has benefits for both performance and failure tolerance. We further presented the interfaces of our framework for both replica and client as well as the interfaces to add further protocols to our toolbox. Moreover, the framework provides an automatic detection of the best composition based on the underlying machine. The policy to select the optimal composition is simple for now, but can be extended in the future.

The implementation of 2PC, Raft and 1Paxos posed several problems. The problems of Raft were too grave and it did not become part of our toolbox. To replicate within a node, we implemented the two simple algorithms based on shared memory (SWMR queue) and message passing (broadcast replication). With the building blocks, we assembled the toolbox to a framework that we showcased by a key value store implementation.

In the last part of this thesis, we evaluated the performance gains from our improvements for a multicore environment. We showed the benefits of changing to an operating system (Barrelfish) that is more suited for a multicore environment. Our framework let us analyse the performance of different compositions of our toolbox. With the help of our framework and a composition of messages passing and shared memory, we were able to scale and replicate a key value

store to 24 cores while maintaining reasonable performance.

9.1 Future Work

9.1.1 Failure Model

If the hardware fails, the machine (or a part of it) is shut down. Our failure model does not handle restarts of hardware and we did not consider replicas rejoining yet. Further, we do not account for software failures that inject false information an try to crash the system. If we want to handle these failures, we have to change the underlying failure model to crash-recovery and handle Byzantine failures.

9.1.2 Protocols

A toolbox can only be as good as its parts. We only implemented 2PC, 1Paxos, broadcast replication, and shared memory as building blocks. The more building blocks, the better we can adapt to the underlying machine. There is a multitude of consensus protocols that are interesting to add to the toolbox. Some examples are the Zookeper atomic broadcast protocol (Zab) [14], Practical Byzantine Fault Tolerance (PBFT) [7], and Viewstamped Replication [22].

9.1.3 Improvements for Multicore

In subsection 4.2.5 we discarded, at least for this thesis, the optimization to adapt the communication topology to the underlying non-uniform message passing latencies. Taking the non-uniform latencies into account, we could further improve the performance of consensus on a multicore machine.

9.1.4 Network Layer

For the beginning we only considered consensus within a single machine. In the future it may be required to not only replicate within a machine but over several machines. The setup we are most interested in is replication within a server rack. To expand the framework to such extend, Remote Direct Memory Access (RDMA) is crucial for a good performance. Unfortunately, Barrelfish still needs some work in this area.

9.1.5 Usage within Barrelfish

There are various examples where our framework could be used within Barrelfish. A simple example is the the System Knowledge Base (SKB) [24] of Barrelfish. The SKB contains information about the system and is centralized but could be distributed with our framework. A more complex problem to tackle that would require some adapting of the framework is the capability system. At the moment, the capability system relies on 2PC. These examples could use our

framework, but at the moment we are only able to run a single instance of the framework. If there is more than one instance, naming collisions can occur and the two instances may interfere with each other.

9.1.6 Framework Cleanup

The framework grew over time from a more benchmark oriented approach. We did not always immediately find a clean solution for implementing parts of our framework. Consequently, there is legacy code that can be replaced with a cleaner solution.

Appendix A

Two Phase Commit

We first implemented 2PC on top of QC-libtask using the discoveries from chapter 4. On Barrelfish we did not only implement 2PC based on a leader, but a more aggressive parallelized version with no leader and the chance of the agreement failing.

A.1 Consensus Inside Message Passing

In section 4.4 we talked about reducing the processing overhead. The main change was the message format itself. Instead of classes we defined the message struct shown in listing A.1.

Listing A.1: 2PC: message struct definition

```
#define REPLY_TAG 1
#define REQUEST_TAG 2
#define PREPARE_TAG 3
#define READY_TAG 4
#define COMMIT_TAG 7
#define ACK_TAG 8

typedef struct{
    uint16_t tag;
    uint16_t size;
    uint16_t msg_size;
    uint16_t client_id;
    uint16_t sender_id;
    uint64_t request_id;
    uint64_t index;
    char* payload;
}message_t;
```

The fields of the struct have the following meaning:

- `Tag:` the tag is the message type.
- `Size:` the size of the message including payload.

- `Msg_size:` only the size of the payload.
- `Client_id:` the id of the client that sent the request.
- `Sender_id:` the id of the replica that the messages was sent from.
- `Request_id:` the request count from the point of view of a client.
- `Index:` the global request count that enables serializability.

The message struct can represent all the message types and a replica can convert the message type by changing the `tag` field. To react to a message, a replica updates its state and changes the `tag` as well as the `sender_id` if necessary. Since we no longer use objects for the messages and removed the C++ library functions, the replica and the client are now written in C. Further, we implemented the tree broadcast as described in section 6.1.

The state of the leader is simply two arrays of counters. The arrays store for each client the number of *ack*s and the number of *ready* messages the leader received. If an agreement is finished, the counters are reset.

A.2 Barrelfish Message Passing

Barrelfish has a lot of support for communication using messages, foremost Flounder [3]. Barrelfish uses waitsets and event handling as basic mechanisms for messages passing. Most of our implementations of the replicas follow the same structure. A Part of the code initializes the state of the replica and sets up the communication channels. The handler functions, which define how the different types of messages are handled, are entered into a struct that is part of the binding process when a communication channel is set up. The signatures of the message handler are defined in a Flounder interface. Moreover, Flounder generates code that simplifies sending and receiving messages and hides some of the complexity. Our Flounder interface definition for 2PC is shown in listing A.2

Listing A.2: 2PC: Flounder interface definition

```
interface tpc "Interface for 2PC protocol" {

  typedef struct {
      uint64 arg1;
      uint64 arg2;
      uint64 arg3;
  } command;

  message request(uint16 client_id,
          uint64 request_id,
          command cmd);

  message prepare(uint16 client_id,
```

```
        uint16 sender_id,
        uint64 request_id,
        command cmd);

message ready(uint16 client_id,
        uint16 sender_id,
        uint64 request_id,
        command cmd);

message commit(uint16 client_id,
        uint16 sender_id,
        uint64 request_id,
        uint64 index,
        command cmd);

message ack(uint16 client_id,
        uint16 sender_id,
        uint64 request_id,
        uint64 index,
        command cmd);

message reply(uint64 request_id);
}
```

The size of the `command` struct can not exceed a certain limit so that a message would be larger than a cache line (for every protocol). The limitation is necessary since the transfer unit of UMP is a cache line. Flounder supports `uint8_t` arrays with undefined length, but sending an array requires a call to `malloc()` which is slow on Barrelfish. Fixing the size of the payload solved the problem. The other message types are according to the 2PC types. There are two additional message types that are not shown and are used for setting up the channels as well as verifying the correctness.

The implementation of the broadcast tree was more difficult than expected because of the UMP channel. If the channel is full, which happens more often when broadcasting along a tree, sending a message returns an error called `FLOUNDER_ERR_TX_BUSY`. To handle a full channel, we add the message arguments and the message type to a queue. Each channel contains a field to store the state of the sender. We used the field to store the state of the message queue. To prevent the same error on the next send, a replica registers for the event in that the channel is free. The registration to the event allows to define a function that should be called when the channel is free as well as the arguments to the function. We implemented a general queue handler function that takes an element of the queue, switches on the type of the element and sends the message accordingly. After successfully sending, the queue handler re-registers itself if there are still elements left in the queue.

A.3 Removing the Leader

In 2PC based on a leader, an agreement can not fail (excluding hardware failures). The leader enforces a total order on the requests and prevents conflicts. By removing the leader and letting the clients directly broadcast *prepare* messages to the replicas, situations where two messages conflict can form. We detect conflicts by treating the first `uint64_t` of the `command` struct as a key. If two clients try to send a *prepare* message for the same key, one of them fails. If a replica receives a new *prepare* message it checks if there is any other agreement ongoing that involves this key. On each replica there is an array that stores the current ongoing agreement key for each client. If the key is not used yet, the key is written into the array and any other incoming *prepare* is aborted. When a replica receives a *commit* message, the array element belonging to said client is reset and can be reused. If a request is aborted, a new message type is needed (listing A.3).

Listing A.3: 2PC: abort message type

```
message abort(uint16 client_id,
        uint64 request_id,
        command cmd);
```

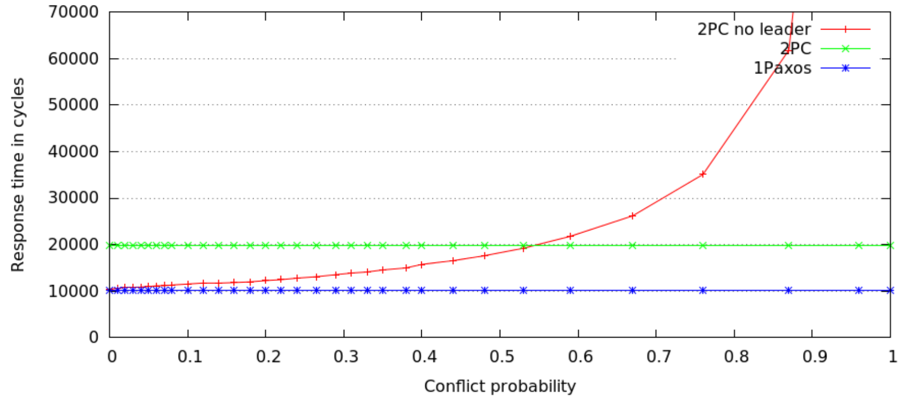
If a client receives a single *abort* message, it informs all the other replicas of the failed request. If a replica receives an *abort* from a client, the replica resets the stored agreement key of that client.

Under these circumstances sequentializability is no longer guaranteed. Instead of a total order over all requests, we can only guarantee that the order of requests that are on the same key are in the same order on all replicas.

A.3.1 Benchmarks

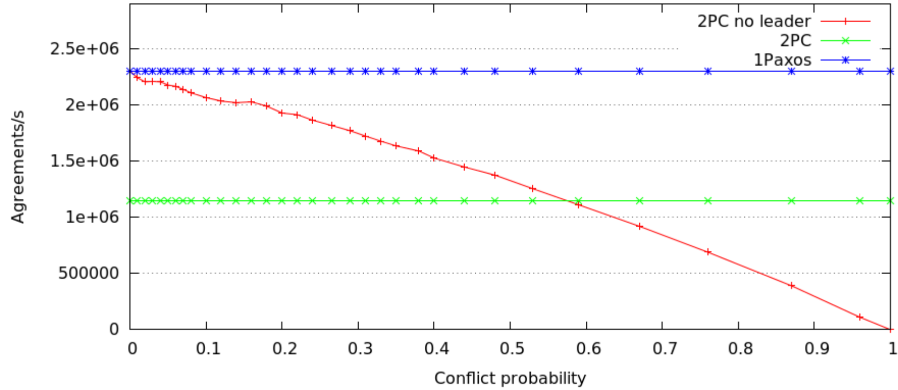
In the main thesis we did not have a look at how 2PC without a leader performs when there are conflicts. A conflict can happen if two agreements are running simultaneously and try to do an operation on the same key. We measured the probability of a conflict by counting the number of aborted requests and compared them to the number of successful requests. The clients send requests with a randomized key. The maximal possible value of a key increased over time. The maximal key started at two and was multiplied by two every minute. The benchmarks are executed on an Ivy Bridge machine with 8 replicas and 10 clients. The result of the benchmark is shown in figures A.1 and A.2.

Figure A.1: 2PC: throughput w.r.t. conflict probability



As the probability of a conflict decreases, the response time reduces. At a collision probability of around 0.55, 2PC without a leader has a lower response time than 2PC. 2PC without a leader does not reach the response time of 1Paxos.

Figure A.2: 2PC: response time w.r.t. conflict probability



2PC without a leader seems to function well even with a conflict probability of 0.1-0.2. To reach a conflict probability of 0.1-0.2 in our setting, around 512 different keys are enough.

Appendix B

Raft

Raft is the only protocol we implemented that is not part of our framework. Raft uses the Flounder interface shown in listing B.1.

Listing B.1: Raft: Flounder interface

```
interface raft "Interface for Raft protocol" {

  message request(uint16 client_id,
                  uint64 value);

  message requestVote_request(
                  uint64 cand_term,
                  uint16 cand_id,
                  uint64 last_log_index,
                  uint64 last_log_term);

  message requestVote_reply(
                  uint64 term,
                  bool granted);

  message appendEntry_request(
                  uint64 term,
                  uint16 leader_id,
                  uint64 prev_log_index,
                  uint64 prev_log_term,
                  uint64 value,
                  uint64 term_to_value,
                  uint64 leader_commit_index,
                  bool empty);

  message appendEntry_reply(
                  uint64 term,
                  uint64 prev_log_index,
                  uint16 replica_id,
                  bool success);


  message reply();
```

```
  message new_leader(uint16 new_leader);

  message setup(uint16 client_id, bool is_replica);
}
```

Raft is the only Flounder interface that does not have the notion of a `command`. We implemented Raft after the basic 2PC and did not think about our framework yet. The code of Raft is looking more like a benchmark than a building block of our toolbox.

The `*_request` and `*_reply` form the two parts of an RPC but are asynchronous. Two asynchronous messages make the implementation for failure safety easier and we did not have to use thread to prevent RPCs from blocking the whole agreement. We added the boolean `empty` to the input parameters of the request message to prevent reserving a value for *Heartbeats*. To the reply parameters we added `prev_log_index` and `replica_id`. The requests and replies are loosely coupled and the leader needs to know from which replica a reply came from as well as for which log entry the *appendEntry_request* messages was sent. The *requestVote* message pair has the same arguments as described in the paper [23].

`FLOUNDER_ERR_TX_BUSY` are handled in the same way as 2PC handles them. Similar to the 2PC implementation, a client can only have one request in the system. The leader replies to a request when a majority of replicas responded to the leader with a successful *appendEntry_reply*. We did not implement a broadcast tree since failures within the tree are hard to fix.

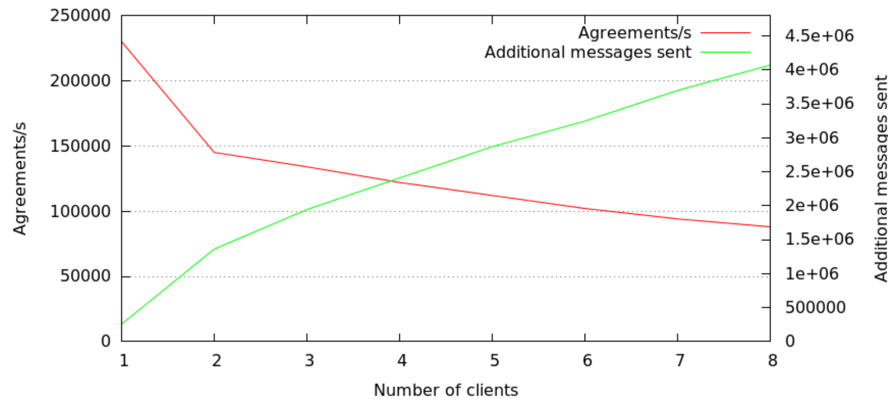
The *requestVote* message pair detects failures of replicas. The frequency of *Heartbeats* is 50ms and an election is started after 250-400ms. A lower timeout time results in false positives and starts a leader election even though the leader is still responsive. If a new leader is elected it informs all clients that it is the new leader by broadcasting a *new_leader* message. Messages are still sent on the channels to a failed replica. Since the channel is never emptied, the queue to a failed replica grows. After the queue reaches a length of 10'000 messages, we assume the replica is dead.

B.1 Problem Benchmarks

We discussed the problem that occurs with our implementation in section 6.2. The problem forms when there are several clients and the gap between the `commitIndex` and the leader's `lastLogIndex` opens up. The leader tries closing the gap by sending more *appendEntry_request* messages to the replicas. Sending unnecessary messages leads to a decrease in performance. To showcase the problem, we benchmarked the Raft implementation on an Ivy Bridge machine. We started 8 replicas and varied the number of clients. Raft uses a log and we

did not implement snapshots/checkpoints to reduce the length of the logs. The log can not grow indefinitely so we limited the number of requests to 100'000 for all client configurations. The throughput as well as the number of additional messages sent (from the leader to any replica) are shown in figure B.1.

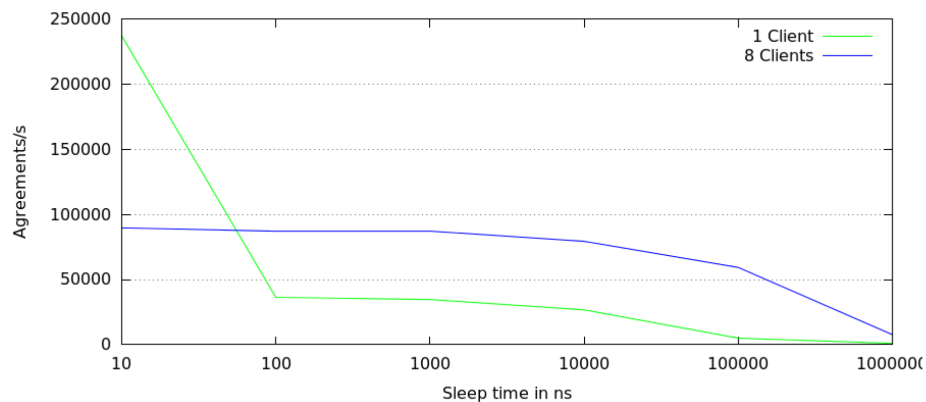
Figure B.1: Raft: throughput and additional number of messages w.r.t. the number of clients



A single client can produce a throughput of well over 250'000 agreements per second. As the number of clients increases to 8, the number of additional messages increases and the throughput reduces. The processing time for additional messages eventually dominates and the processing time left for useful messages reduces leading to a lower throughput.

To prevent the gap between `lastLogIndex` and `commitIndex` to open, we added a sleep time between requests. The effect of the sleep time is shown in figure B.2.

Figure B.2: Raft: throughput using 1 and 8 clients with a varying sleep time

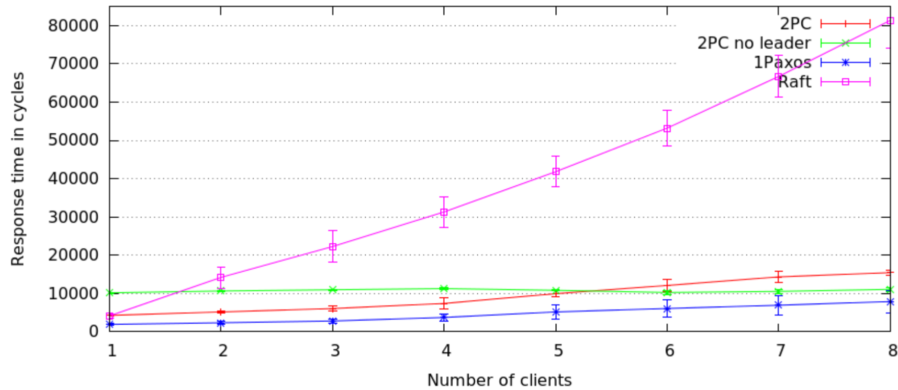


If the sleep time is increased (**Note logarithmic scale of sleep time**), the replicas have enough time to keep up with the leader and close the gap to its `lastLogIndex`. Even 10ns sleep time between requests is enough to mitigate the problem so that 8 clients have a higher throughput than a single client. As the sleep time increases the difference between 1 and 8 clients slowly gets to the point where the throughput of 8 clients equals the throughput of a single client times eight.

B.2 Response Time

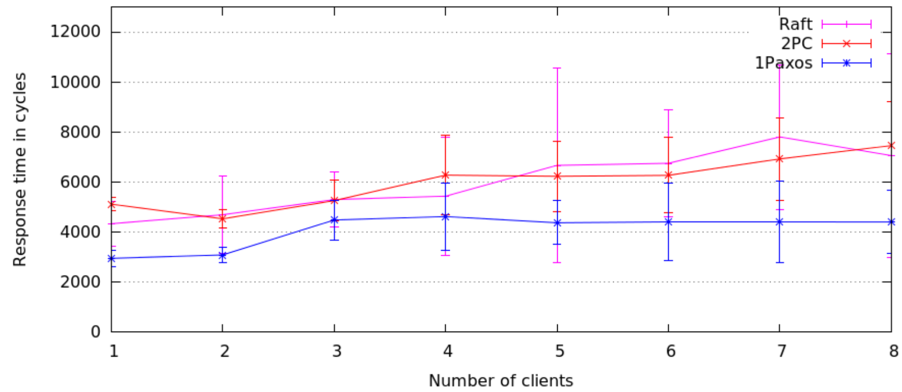
To show how Raft stacked up against the other protocols, we did some benchmarking. We tested using 8 replicas with a varying number of clients. The setup is the same as in subsection 7.2.2 (Ivy bridge machine). For Raft we limited the measurement to 1'000'000 requests instead of the 20 seconds interval. The results are shown in figure B.3.

Figure B.3: Raft: response time w.r.t. number of clients



As the load increases, the response time of Raft gets out of hand since the number of agreements per second Raft can handle decreases. The more interesting case is when we introduce a sleep time on the clients of around 100 μs. The coherency of the logs of the leader and the replicas is well maintained since the number of additional messages is less than 500 for all configurations. In the lower load setting, Raft works better as shown in figure B.4.

Figure B.4: Raft: response time with sleep time between requests w.r.t. number of clients



The response time suggests that Raft has a similar performance as the standard 2PC. For a protocol that can tolerate failures the response time is acceptable. Still, the problems at a higher load can only be solved if we rely on normal RPCs as well as introducing more threads. Changing the implementation certainly would increase the response time by a large margin.

Appendix C

1Paxos

The Flounder interface of 1Paxos is shown in listing C.1.

Listing C.1: 1Paxos: Flounder interface

```
interface onepaxos "Interface for 1Paxos protocol" {

  typedef struct {
      uint64 arg1;
      uint64 arg2;
      uint64 arg3;
  } command;

  message request(uint16 client_id,
                  uint64 request_id,
                  command cmd);
  message reply();

  message prepare(uint16 client_id,
                  uint64 request_id,
                  uint64 index,
                  uint64 n,
                  command cmd);

  message prepare_response(uint16 sender_id,
                  uint64 index,
                  uint64 n);

  message accept(uint16 client_id,
                 uint64 request_id,
                 uint64 index,
                 uint64 n,
                 command cmd);

  message learn(uint16 client_id,
                uint64 request_id,
                uint64 index,
                uint64 n,
                command cmd);
```

```
  message abandon(uint16 sender_id,
                  uint16 leader_id,
                  uint64 n);

   // Acceptor fail
  message is_current_leader(uint16 sender_id);
  message is_current_leader_response(bool success);

  // Leader fail
  message get_current_acceptor();
  message get_current_acceptor_response(uint16 id);
  message new_leader(uint16 new_leader,
                     uint64 next_rid);

  // Changeing acceptor/leader
  message change_key_figure(uint16 new_id,
                            uint16 sender_id,
                            bool for_leader);

  // detection of leader/acceptor fail
  message is_alive();
  message is_alive_response(uint16 sender_id);
}
```

The messages *prepare*, *prepare response*, *accept*, and *learn* have the same purpose as in basic Paxos. When a client sends a *request* to the leader, the leader sends an *accept* message to the acceptor. The acceptor, depending on the proposal number N (used similar to the term in Raft), broadcasts either a *learn* to all replicas or an *abandon* message to the leader. If the leader receives an *abandon* message, it will no longer act in the role of the leader.

A failure is detected by *is_alive* and *is_alive_response* messages. The messages are sent periodically from the leader to the acceptor to detect acceptor failures. Further, they are sent from all other replicas to the leader. The period between messages is 250 ms. The periodic events are started by the function `periodic_event_create()`. Additionally to sending a message, the code checks if a response to the previous message was received. If not, a new leader/acceptor election is started. The detection of a response contains any *learn* messages received to prevent false positives in the case of high load.

Once a failure of the acceptor is detected, the leader broadcasts an *is_current _leader* to all other replicas. If the leader receives *is_current_leader_response* from a majority of nodes, it proceeds further. In how we choose a new acceptor we differ from the implementation proposed in the paper. Instead of using a "utility" Paxos we let the leader decide on the new acceptor. After choosing a new acceptor, the leader broadcasts a *change_key_figure* message to all the replicas to announce the change in acceptor. Next, the leader sends a *prepare* message to the new acceptor with the last value in the proposal list. The leader keeps track of the requests that are not yet accepted in a linked list. When the leader receives a request, it is added to the list. When the leader receives

a corresponding *learn* message, the length of the list is checked. The leader keeps track of how many requests it received from clients as well as the number of *learn* messages it received. If the length of the queue is larger than the difference between the number of requests and the number of *learn* messages, it is safe to dequeue the oldest request in the queue. The order of requests is the same as the order of *learn* messages since the channels between the replicas offer the FIFO property. When the new acceptor receives the *prepare* message, it increases its highest proposal number and responds with a *prepare response* message. The leader on its turn sends all messages that it has in its request list as an *accept* message to the acceptor. By storing all the requests in a list, we do not have to inform the clients to resend their last request and still maintain consistency of the replicas. If a replica receives a *learn* message that it has already committed, it can detect the duplicate by the `request_id` and the `client_id`.

A leader failure is detected by the replicas through *is_alive* messages. Here the *is_alive* messages are sent every 350 ms with an additional randomized backoff of up to 150 ms. When the time between messages exceeds the timeout, the replica tries to become leader and broadcasts a *get_current_acceptor()* message. When a replica responds with the acceptor id, it is counted as a vote for the new leader similar to Raft. Any other replica that should try to get a vote from this replica, does not get it. When the candidate replica receives the same id from a majority of nodes, it can proceed and send a *change_key_figure* message. After the change, the new leader sends a *prepare* message to the acceptor and the same procedure as previously starts. The old leader may have received *request* messages that it did not send further to the acceptor before failing. To prevent inconsistencies, the clients are informed by *new_leader* messages to resend their last request to the new leader.

Bibliography

[1] D. Alistarh, R. Guerraoui, P. Kuznetsov, and G. Losa. On the cost of composing shared-memory algorithms. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, pages 298–307, New York, NY, USA, 2012. ACM.

[2] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, SOSP '91, pages 95–109, New York, NY, USA, 1991. ACM.

[3] A. Baumann. Technical note 011: Inter-dispatcher communication in barrelfish, 2011.

[4] A. Baumann, P. Barham, P.-E. Dagand, T. Harris, R. Isaacs, S. Peter, T. Roscoe, A. Schüpbach, and A. Singhania. The multikernel: A new os architecture for scalable multicore systems. In *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, SOSP '09, pages 29–44, New York, NY, USA, 2009. ACM.

[5] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. Lightweight remote procedure call. *ACM Trans. Comput. Syst.*, 8(1):37–55, Feb. 1990.

[6] B. N. Bershad, T. E. Anderson, E. D. Lazowska, and H. M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(2):175–198, May 1991.

[7] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems (TOCS)*, 20(4):398–461, Nov. 2002.

[8] P.-E. Dagand, A. Baumann, and T. Roscoe. Filet-o-fish: Practical and dependable domain-specific languages for os development. In *Proceedings of the Fifth Workshop on Programming Languages and Operating Systems*, PLOS '09, pages 5:1–5:5, New York, NY, USA, 2009. ACM.

[9] T. David, R. Guerraoui, and M. Yabandeh. Consensus inside. In *Proceedings of the 15th International Middleware Conference*, Middleware '14, pages 145–156, New York, NY, USA, 2014. ACM.

[10] C. Dwork, N. A. Lynch, and L. J. Stockmeyer. Consensus in the presence of partial synchrony. *J. ACM*, 35(2):288–323, 1988.

[11] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: An operating system architecture for application-level resource management. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 251–266, New York, NY, USA, 1995. ACM.

[12] L. Fang, A. D. Sarma, C. Yu, and P. Bohannon. Rex: Explaining relationships between entity pairs. *Proc. VLDB Endow.*, 5(3):241–252, Nov. 2011.

[13] E. Gafni and L. Lamport. Disk paxos. *Distrib. Comput.*, 16(1):1–20, Feb. 2003.

[14] F. P. Junqueira, B. C. Reed, and M. Serafini. Zab: High-performance broadcast for primary-backup systems. In *DSN*, pages 245–256. IEEE, 2011.

[15] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, May 1998.

[16] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, October 2006.

[17] L. Lamport and M. Massa. Cheap paxos. In *DSN '04: Proceedings of the 2004 International Conference on Dependable Systems and Networks (DSN'04)*. IEEE Computer Society, 2004.

[18] H. C. Lauer and R. M. Needham. On the duality of operating system structures. *Operating Systems Review*, 13(2):3–19, 1979.

[19] H. M. Levy. *Capability-Based Computer Systems*. Butterworth-Heinemann, Newton, MA, USA, 1984.

[20] J. Liedtke. On micro-kernel construction. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*, SOSP '95, pages 237–250, New York, NY, USA, 1995. ACM.

[21] R. Nishtala and K. A. Yelick. Optimizing collective communication on multicores. In *Proceedings of the First USENIX Conference on Hot Topics in Parallelism*, HotPar'09, pages 18–18, Berkeley, CA, USA, 2009. USENIX Association.

[22] B. Oki and B. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing (PODC)*. ACM, Aug. 1988.

[23] D. Ongaro and J. Ousterhout. In search of an understandable consensus algorithm. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 305–319, Philadelphia, PA, June 2014. USENIX Association.

[24] A. Schuepbach, S. Peter, A. Baumann, T. Roscoe, P. Barham, T. Harris, and R. Isaacs. Embracing diversity in the barrelfish manycore operating system. In *In Proceedings of the Workshop on Managed Many-Core Systems*, 2008.

ETH

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Consensus on a multicore machine

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	First name(s):
Häcki	Roni

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date
Stans, 02.09.2015

Signature(s)
Roni Häcki

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.