Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Distributed Computing

Implementing a Distributed Reliable Database

Bachelor Thesis

Florian Morath

fmorath@ethz.ch

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Conrad Burchert
Prof. Dr. Roger Wattenhofer

February 22, 2018

Acknowledgements

I want to thank Conrad Burchert, who guided me through this thesis, gave me great advice on many questions and who was always passionate about the topics of our meetings.

Abstract

In this thesis, we implement a fault-tolerant distributed state machine called piChain and make it available to others as a library. piChain inherits features of a blockchain, providing eventual consistency, and of Paxos, providing strong consistency.
The library is compared with current implementations of a popular consensus algorithm called Raft. A distributed database is built on top of the library.

Contents

Chapter 1

Introduction

1.1 Motivation

piChain [1] is a proposal of a new fault-tolerant distributed state machine. It inherits features of Paxos [2, 3], providing strong consistency, and of a blockchain [4], providing eventual consistency. A blockchain is a fault-tolerant data structure that is used as a core component in many cryptocurrencies where it has proven itself for its scalability and simplicity. The underlying data structure of piChain is a blockchain which organizes the transactions. Since the main disadvantage of a blockchain is that it is only eventually consistent, piChain additionally uses Paxos, a strongly consistent consensus algorithm.

The focus of the piChain design was put on the following properties:

- **Fault-Tolerance**: piChain can handle crashes, crash-recoveries, message omissions and network partitions.
- **Speed and Simplicity**: piChain is a light protocol which should be simple to understand. It has no heavy weight leader election subroutine. In a healthy state, in which the system usually is, strong consistency is achieved in one message round-trip time.
- **Quietness**: There is no heartbeat needed, meaning that if no new transactions are created, no messages are sent.
- **Scalability**: The number of piChain nodes can be scaled up to hundreds of nodes distributed around the globe. There is no subroutine that produces a quadratic number of messages.

A distributed fault-tolerant state machine has many applications. It can be used whenever a distributed synchronization mechanism is needed in an environment where nodes may crash. Use cases include the following:

- An electronic payment system where financial transactions are part of the consistent state.

- A distributed storage system where executable commands are part of the consistent state.
- A distributed lock manager that organizes access to resources where the locks on the resources are part of the consistent state, e.g., Chubby [5].

The goal of this thesis is to implement the proposed algorithm and make it available to others as a library. Also a simple distributed database is implemented as an application of the library to show how the library can be used and that it works.

1.2 Related Work

There exists a number of implementations that try to solve the same problem. To be able to make a meaningful comparison regarding their performance, I only consider implementations that use the same programming language as is used for this thesis, namely the Python programming language.

The most recent champion among consensus algorithms is probably Raft [6]. Raft's core component is an explicit leader election strategy. The leader is responsible for the log replication and regularly informs other nodes about its existence with so called heartbeat messages. If nodes do not receive heartbeat messages, they timeout and start a new leader election.
Comparing this algorithm to piChain, one can clearly observe that simplicity is an important part of the design of both algorithms. Where the algorithms differ is the leader election. While in Raft leaders are elected explicitly, in piChain a node can promote and demote itself without communication, which makes it a light algorithm.

The most popular Python implementations of Raft are *PySyncObj*[1] and *raftos*[2]. In the following, I look at how networking and data serialization are approached in both implementations, which I consider the most important parts of a replicated state machine.
PySyncObj uses TCP connections to establish a communication channel. It does not use any high-level networking library, instead it builds the connection management from scratch using the Python *socket*[3] module. While this gives you more control, using a high-level networking library usually saves you time because you can rely on something that has been used and tested. In addition, it decreases maintenance work. That is why for this thesis a popular networking library called *Twisted*[4] is used. For data serialization, *PySyncObj* mainly uses

[1] `https://github.com/bakwc/PySyncObj`
[2] `https://github.com/zhebrak/raftos`
[3] `https://docs.python.org/3/library/socket.html`
[4] `https://twistedmatrix.com/trac/`

the Python *pickle*[5] module. Pickling is the process whereby a Python object hierarchy is converted into a byte stream such that it can be either sent over the network or written to disk. The advantage of it is that it can serialize any Python object without having to add any extra code. There are problems however with interoperability, security and performance, which is the reason *pickle* is not used in this thesis.
Raftos uses the *asyncio*[6] module for its networking part. Data is sent using the integrated UDP protocol. *Asyncio* was added to Python in version 3.4 and was heavily inspired by *Twisted*. *Twisted* is probably one of the oldest libraries that support asynchronous network programming in Python and has been used by many people. Raftos uses *MessagePack*[7] for data serialization, which is an efficient binary serialization format. For this thesis a data serialization module called *CBOR*[8] is used since it performs slightly better than *MessagePack*.

[5]https://docs.python.org/3/library/pickle.html
[6]https://docs.python.org/3/library/asyncio.html
[7]https://msgpack.org/
[8]http://cbor.io/

Chapter 2

State Replication

In this chapter the theory of the piChain algorithm is illustrated. The necessary background knowledge is given in the following two sections. Note that this chapter is based on [1, 8].

2.1 Basics

piChain is a replicated state machine. Such a state machine has some internal state and typically receives requests from clients, processes them and responds back to the clients. An example of a state machine is a storage system that receives SQL queries.
A replicated state machine [7] consists of multiple servers, each running an instance of the state machine. It is important that each server executes the same set of commands in the same order. Execution needs to happen deterministically, else different servers may have inconsistent states. The consensus algorithm ensures that commands are executed in the same order on all servers. This leads to state replication because of deterministic execution.

2.2 Paxos Theory

The Paxos algorithm is a core component of piChain. It was invented by Leslie Lamport.
Each instance of the Paxos protocol reaches agreement on a single command. Algorithms that implement a repeated version of Paxos, i.e., ones that combine multiple Paxos instances to achieve state replication, are often called multi-Paxos algorithms. piChain can be seen as a particular instance of a multi-Paxos algorithm. The Paxos algorithm can be seen in Figure 2.1.

```
Client (Proposer)                              Server (Acceptor)

Initialization ...............................................................

$c$        ◁ command to execute                $T_{\max} = 0$    ◁ largest issued ticket
$t = 0$    ◁ ticket number to try
                                               $C = \bot$        ◁ stored command
                                               $T_{\text{store}} = 0$  ◁ ticket used to store $C$

Phase 1 ......................................................................

 1: $t = t + 1$
 2: Ask all servers for ticket $t$
                                                3: if $t > T_{\max}$ then
                                                4:    $T_{\max} = t$
                                                5:    Answer with ok($T_{\text{store}}$, $C$)
                                                6: end if

Phase 2 ......................................................................

 7: if a majority answers ok then
 8:    Pick ($T_{\text{store}}$, $C$) with largest $T_{\text{store}}$
 9:    if $T_{\text{store}} > 0$ then
10:       $c = C$
11:    end if
12:    Send propose($t$, $c$) to same majority
13: end if
                                               14: if $t = T_{\max}$ then
                                               15:    $C = c$
                                               16:    $T_{\text{store}} = t$
                                               17:    Answer success
                                               18: end if

Phase 3 ......................................................................

19: if a majority answers success then
20:    Send execute($c$) to every server
21: end if
```

Figure 2.1: The Paxos algorithm. Figure taken from [8].

First, one needs to understand the concept of a ticket. The Paxos algorithm uses tickets, which are a weaker form of a lock. A server can issue a ticket for its resources even if previously issued tickets have not been returned. A server accepts a proposal from a client only if the ticket is still valid. In the Paxos algorithm, instead of the servers choosing the ticket numbers, the clients suggest it. This ensures that proposals from clients are associated with a unique ticket number.

Paxos consists of three phases:

Phase 1: A client asks all servers for a ticket by suggesting a ticket number. Each server stores its currently largest issued ticket number to keep track of valid tickets. The servers accepting the suggestion now consider the ticket number as valid and also provide the client with their currently accepted proposal if there is any. A proposal consists of a command and a ticket number that is associated with it.
Phase 2: The client can propose a command as soon as a majority of servers issued a ticket. If any of the servers that replied has already accepted a proposal, the client supports the proposed command with the largest associated ticket number, instead of his own command. Because a ticket may get invalidated by a client requesting a new ticket, on receiving a proposal the server first checks if the associated ticket is still valid. If it is valid, the proposal is accepted and stored. The server acknowledges this to the client.
Phase 3: If a majority of servers have accepted the proposal, the client tells every server to execute the proposed command.

Claim 2.1. This protocol ensures that every proposal p_2 that is made after an already accepted proposal p_1 that is stored by a majority of servers and that proposed command c proposes the same command c.

Proof. Assume p_1 has ticket number t_1 and p_2 has ticket number t_2 such that $t_1 < t_2$ and t_2 is the smallest among all proposals with ticket number greater than t_1. Since a majority of servers stored p_1, the client with proposal p_2 is informed about p_1 during the request of ticket t_2 since $t_2 > t_1$. Proposal p_2 supports the command proposed with the largest ticket number, which is c. Assume there exists a proposal p_3 with command $c_3 \neq c$ and ticket number $t_3 > t_1$. This would imply that $t_3 < t_2$, else a client proposing p_2 would not know about c_3. This contradicts the assumption that t_2 is the smallest among all proposals with ticket number greater than t_1. □

2.3 piChain Theory

Assume there exists a system that consists of a collection of piChain nodes. The goal of piChain is that every node executes transactions in the same order. Transactions can for example be commands in a storage system or financial transactions. A transaction consists of two fields:

- A content field which is set by the application using piChain.
- An ID which is a combination of the ID of the piChain node that created the transaction (called creator ID) and a sequence number to make it unique.

A transaction, once created, is broadcast to all other nodes.
The underlying data structure that stores the transactions is a tree of blocks. A block consists of three fields:

- A list of transactions.
- An ID which is again a combination of the creator ID and a sequence number.
- A pointer to a parent block.

Blocks are created by arbitrary nodes in the system. The parent of a newly created block is the deepest block the creator has seen, where the depth of a block is the sum of the depth of its parent and the number of transactions the block contains. To break ties, the block ID is used. Blocks are also broadcast to all other nodes once created. The root block (called Genesis block) is hard-coded and thus known to every node.

If every node could simply create a block once it sees a new transaction, this would lead to a lot of forks and reorderings of the blockchain. The blockchain, the longest path from the root to a leaf, represents the consistent state among all nodes. That's why nodes are associated with states they can be in. Each node is either quick, medium or slow.

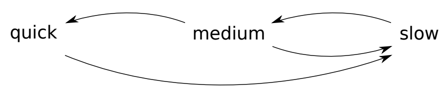

Figure 2.2: The state transitions. Figure taken from [1].

- **quick**: If a quick node sees a transaction that it has not yet seen (either from another node or itself), it immediately[1] creates a block and includes the transaction.
- **medium**: If a medium node sees a new transaction, it gives the quick node enough time to send it inside a block. If after this time the transaction was not received yet, the medium node assumes that there is no quick node and it creates a block, including the transaction, itself. This promotes the medium node to the quick state. The amount of time the medium node waits is equal to $(1 + \epsilon)$ RTT's, since if the medium node creates the transaction, it has to be sent to the quick node and the block, including the transaction, has to be sent back to the medium node.

[1]The quick node can also wait a predefined accumulation time to gather multiple transactions.

- **slow**: If a slow node sees a new transaction, it gives the quick and medium nodes time to send it inside a block and if they do not, it creates a block including the transaction. This promotes the node to the medium state. A slow node has to wait $(2 + \epsilon)$ RTT's, since it needs to wait 1 RRT longer than the medium nodes to give them a chance to create a block.

During normal operation, the nodes are in a so called healthy state, meaning that there exists exactly one quick node, the others being in a slow state. If a quick node crashes, slow nodes compete to get promoted. A random duration is added to their waiting time, such that only one slow node creates a block in expectation. The medium state is only a transient state which is used to get to the quick state. A node demotes itself to slow if either of two scenarios happen:

- Receiving a block that was created by a quick node. This means there already exists a quick node but we want to avoid having multiple quick nodes in the system.
- Receiving a block that is the new deepest block. If multiple slow nodes compete and create a block, only one should win and stay medium. The winner is the one with the deeper block.

A blockchain still only provides eventual consistency. We want to be able to say that certain blocks remain in their place forever, i.e., that all predecessor blocks up to the root do not change anymore. Such blocks are called committed. How can we ensure that committed blocks are final and cannot be uncommitted again?
This is where Paxos is used. Whenever a quick node creates a block and is not in the process of committing another block, it tries to commit this block by initiating the Paxos algorithm. A committable block must be a descendant of all previously committed blocks. Committing a block commits all the transactions in that block, and all its predecessor blocks[2] up to the root. The Paxos algorithm ensures that if there are multiple quick nodes[3] trying to commit a block, that then there is agreement among the nodes which block will be committed. The Paxos algorithm adjusted to work with blocks can be seen in Figure 2.3.

[2]Many of those blocks may already be committed.
[3]Only quick nodes are allowed to initiate the Paxos algorithm.

```
Quick Node                                      All Nodes
$b_{\text{prop}}$ = deepest block               $b_{\text{max}} = \bot$
$b_{\text{com}} = \bot$                         $b_{\text{prop}} = \bot$
                                                $b_{\text{supp}} = \bot$

Phase 1 ........................................................................
1:  Send try($b_{\text{new}}$) to all nodes
                                                2:  On receiving a try($b_{\text{new}}$) message:
                                                3:  if $b_{\text{new}}$ deeper than $b_{\text{max}}$ then
                                                4:      $b_{\text{max}} = b_{\text{new}}$
                                                5:      Answer with ok($b_{\text{prop}}$, $b_{\text{supp}}$)
                                                6:  end if
Phase 2 ........................................................................
7:  Majority responded with ok($b_{\text{prop}}$, $b_{\text{supp}}$):
8:  $b_{\text{com}} = b_{\text{new}}$
9:  if some response included $b_{\text{prop}} \neq \bot$ then
10:     $b_{\text{com}} = b_{\text{prop}}$ with deepest $b_{\text{supp}}$
11: end if
12: Send propose($b_{\text{com}}$, $b_{\text{new}}$) to all nodes
                                                13: On receiving a propose($b_{\text{com}}$, $b_{\text{new}}$) message:
                                                14: if $b_{\text{new}} = b_{\text{max}}$ then
                                                15:     $b_{\text{prop}} = b_{\text{com}}$
                                                16:     $b_{\text{supp}} = b_{\text{new}}$
                                                17:     Answer with ack($b_{\text{com}}$)
                                                18: end if
Phase 3 ........................................................................
19: Majority responded with ack($b_{\text{com}}$):
20: Send commit($b_{\text{com}}$) to all nodes
```

Figure 2.3: Committing: Paxos with Blocks. Figure taken from [8].

First, note that a node can be both a Paxos server and a Paxos client at the same time. b_{new} is the block that the quick node wants to commit. Every node associates each instance of Paxos with three variables:

- $\mathbf{b_{max}}$: The deepest b_{new} that the node has seen in Phase 1. Since the depth of b_{new} acts as the suggested ticket number of the quick node, the depth of b_{max} corresponds to the largest issued ticket number T_{max} in Paxos.
- $\mathbf{b_{prop}}$ and $\mathbf{b_{supp}}$: b_{prop} is the proposed block to be committed and b_{supp} is the block supporting b_{prop}, i.e., the ticket associated with b_{prop}. Once a quick node is allowed to make its own proposal, it supports the b_{prop} with the deepest associated b_{supp}. (b_{prop}, b_{supp}) is like the accepted proposal

(T_{store}, C)[4] that may be sent back from the server after a successful ticket request in Phase 1.

Apart from those three variables, the adjusted Paxos algorithm is exactly the same as the original illustrated in Figure 2.1.

[4]see Figure 2.1.

Chapter 3

Implementation

This chapter is about the implementation of the piChain package[1]. We elaborate on major challenges that arose.

3.1 Interface Design

The interface should basically consist of two methods:

- A method that creates a transaction. One must be able to pass the content of the transaction as an argument.
- A method that can be implemented by the application using the API and is called once a transaction has been committed.

An illustration of how the package can be used:

```python
from piChain import Node

def tx_committed(commands):
    """
    Args:
        commands (list of str): List of committed Transaction
            commands.
    """
    for command in commands:
        print('command committed: ', command)

def main():
    node_index = 0
    peers = {
        '0': {'ip': '127.0.0.1', 'port': 7980},
        '1': {'ip': '127.0.0.1', 'port': 7981},
        '2': {'ip': '127.0.0.1', 'port': 7982}
    }
    node = Node(node_index, peers)
```

[1]https://github.com/florianmorath/piChain

```
    node.tx_committed = tx_committed
    node.start_server()

    node.make_txn('sql query')
```

First, one needs to setup the *Node* instance. A *Node* constructor takes two arguments, a *node_index* and a *peers* dictionary. The *peers* dictionary contains an (IP, port) pair for each node. With the *node_index* argument one can select which node from the *peers* dictionary is running locally. The *tx_committed* field of a *Node* instance is a callback that is called once a transaction has been committed. By calling *start_server()* on a *Node* instance, the local node tries to connect to its peers and listens for incoming messages. Transactions can be committed by calling *make_txn()* on the *Node* instance, passing the transactions content as an argument.

3.2 Architecture

The piChain package consists of the following modules:

- **Logic module**: This module defines the logic of the piChain protocol. It implements the *Node* class which represents a piChain node and specifies how a piChain node should behave. Most importantly, this includes how a piChain node should respond to a received Paxos message, a transaction and a block. This module includes no networking functionality at all, which instead is separated into the networking module.
- **Networking module**: This module implements the networking functionality between piChain nodes. This mainly includes establishing a peer-to-peer network between nodes and sending and receiving byte streams to and from other nodes. The networking library used is called *Twisted*. There is a *Connection* class which represents a connection between two nodes and implements a communication protocol, and a *ConnectionManager* class which keeps track of *Connection* instances.
- **Blocktree module**: This module implements the *Blocktree* class which manages a tree of blocks. It keeps track of special blocks, such as the last committed block and the head block[2] of the tree. The module provides operations like adding blocks and checking for the validity of new blocks.
- **Message module**: This module defines the representation of all objects that need to be sent over the network such as a block, a transaction and a Paxos message. Each class provides functionality to serialize an instance

[2]That is the deepest block in the block tree respectively the head of the blockchain.

of itself, i.e., converting the object to a sequence of bytes, and to revert the process, i.e., to deserialize.

- **Config module**: This is the module where all the settings of the piChain package can be adjusted such as the accumulation time of a quick node and debugging settings.

The relation between the modules can be seen in Figure 3.1 below. The networking and logic modules are the main modules whereas the message, blocktree and config modules are helper modules which are used by the two main modules. The networking and logic modules are related through a subclass relation, namely the *Node* class in the logic module is a subclass of the *ConnectionManager* class in the networking module. This allows the *Node* class to directly call networking methods like *broadcast*, implemented in the *ConnectionManager*, and also allows it to override and implement the abstract receive-methods in the *ConnectionManager*, which the *ConnectionManager* calls based on the message type of incoming messages. This setup admits a clear separation between networking functionality and the piChain protocol which facilitates development and maintenance.
Also each *Node* instance contains a field that is an instance of the *Blocktree* class.

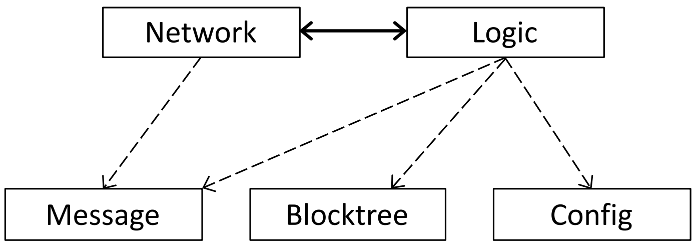

Figure 3.1: Relation between different modules.

3.3 Transactions and Blocks

3.3.1 Transaction Reception

A reception of a transaction is handled in the *Node* class. Once a node receives a transaction, it first checks whether it already has seen this transaction or not. This is done with a set called *known_txs* that contains the transaction IDs of all transactions seen so far (including the transactions that are contained in blocks). The set data structure in Python has the advantage of a constant time complexity on average for the lookup, insert and delete operations since it is implemented as a hash table.
If the transaction has not yet been seen, the node must start a timeout and

somehow remember the transaction. This is done as follows:
The transaction is appended to a list called *new_txs*. This list contains all received transactions that are not yet seen included in a block in the order in which they are received. If a transaction is appended and it is the first element in the list, a timeout, whose length depends on the node state, is started. We call this transaction that initiated the timeout the oldest transaction (*oldest_txn*). Once the timeout is over, the node checks if the oldest transaction is still in the *new_txs* list. If yes, a block including all the transactions in the *new_txs* list is created[3], then broadcast and tried to be committed if the node is a quick node.
There are two scenarios in which the *new_txs* list may change:

- If a block is created: *new_txs* is emptied because the new block contains the transactions inside the *new_txs* list.
- If a block is added to the block tree as the new head block: Transactions included in a new block are removed from the *new_txs* list.

In such a scenario where the *new_txs* list changes, the oldest transaction which initiated the timeout may be removed. If the list is empty after this, we are good, but if there are still other elements in the list, a new timeout has to be started with the first element in the list being the new oldest transaction.

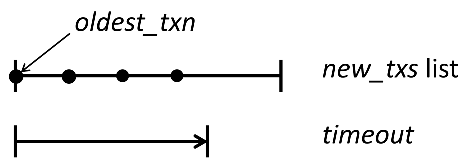

Figure 3.2: Timeout handling of new transactions. The dots in the *new_txs* list represent transactions where the *oldest_txn* is the first item in the list. The arrow on the bottom shows a possible duration of a timeout. Time flows from left to right and the timeout starts on the reception of the *oldest_txn*. Before the timeout other transactions may be added to the *new_txs* list.

3.3.2 Blocktree Module and Block Reception

The blocktree module consists of a hard-coded genesis block (the root of the blockchain), which contains no transactions and has no parent block, and the *Blocktree* class. The *Blocktree* class represents a tree of blocks and the most important fields are:

- **head_block**: The head block is the deepest block in the block tree and thus the head of the blockchain.

[3] A creation of a block promotes a node.

- **committed_block**: The last committed block. It implicitly defines all other committed blocks, namely its predecessors up to the root.
- **nodes**: This is a dictionary containing all the blocks in the block tree. It maps from a block ID to a block.

The *Blocktree* class provides functionality to add blocks, check the validity of a block and to find out the relationship between two blocks.
A reception of a block is handled in the *Node* class. Once a node receives a block, it first checks if it has to demote itself to slow, which is the case if the creator of that block is a quick node[4] or if the block is the deepest block seen so far.
After that, the node checks if it is a valid block. A valid block must be a descendant of the last committed block (*committed_block*) and must be the deepest block seen so far. An invalid block is rejected and a valid block becomes the new head block.

3.3.3 Blockchain Forks

We call the situation where the new head block is not a descendant of the current head block a fork. A fork happens if two different blocks are created that have the same parent block as illustrated in Figure 3.3. This mainly happens in the following two scenarios:

- The single quick node crashes and multiple slow nodes eventually timeout and create a block more or less simultaneously. But since only one of the slow nodes stays medium, the system could quickly converge to a healthy state with a single quick node and resolve the fork. This scenario is rather unlikely to happen, since only one slow node creates a block in expectation.
- A network partition: A partition divides the nodes into two subgroups that cannot communicate anymore. An example of a partition can be seen in Figure 3.4. Both sides of a partition eventually have a single quick node. The two quick nodes create blocks independently from each other, i.e., at one block the blockchain forks and both nodes extend a different side of the fork as illustrated in Figure 3.3. The fork can be resolved as soon as communication between the two sides is established again.

[4]A block includes a field that informs about the state of its creator (*creator_state*).

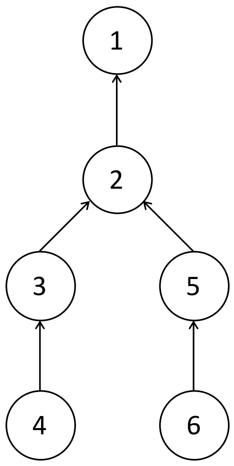

Figure 3.3: A blockchain fork. Block 1 is the genesis block where the blockchain starts and at block 2 it forks, i.e., block 2 has two different children.

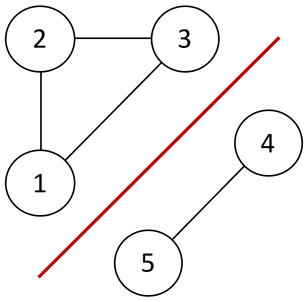

Figure 3.4: A network partition. Node 1, 2 and 3 are separated from node 4 and 5.

There is a method called *move_to_block* inside the *Node* class which sets the argument block to be the new head block. The block reorganization functionality is implemented as part of this method. The situation is illustrated in Figure 3.5. There is the current head block, the target block, which is the block that will be the new head block, and the common ancestor of those two blocks. We basically need to broadcast the transactions that will not be on the path from the genesis block to the target block anymore in order to not lose them. To do this we go over all blocks on the path from the current head block to the common ancestor and add the transactions included in those blocks to a set (called *to_broadcast*). Then we go over the blocks on the path from the common ancestor to the target block and remove the transactions included in those blocks from the *to_broadcast* set (if included) to avoid committing transactions twice. All transactions in the *to_broadcast* set are broadcast to the other nodes and the target block is set to be the new head block.

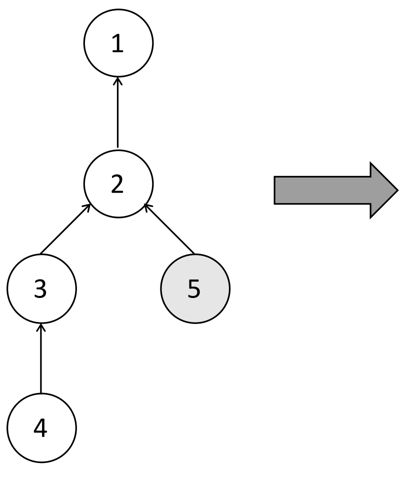

Figure 3.5: A reorganization of the blockchain. On the left we see a blockchain that forks at block 2. Block 5 is the current head block and block 4, called the target block, will be the new head block. On the right we see the blockchain after the reorganization, during which block 5 has been removed.

3.3.4 Missing Blocks

One possibility to miss a block is if a node crashes, since during the time the node is offline it does not receive any messages, including blocks, from other nodes. Another possibility is during a network partition, since there is no communication possible between the two sides of a partition. A third possibility is if messages including a block are lost.
After a node reestablished connection to other nodes, it may request missing blocks from them by using the following two messages implemented by the message module:

- ***RequestBlockMessage***: This message contains the block ID of the missing block. A node broadcasts it if it detects a missing block.
- ***RespondBlockMessage***: This message contains a list of blocks. If a node receives a *RequestBlockMessage*, it checks if it has the missing block. If yes, it sends it included in a *RespondBlockMessage* to the node missing

the block. The node also includes a predefined number[5] of ancestor blocks to allow faster recovery in case multiple blocks are missing.

Whenever a node encounters a block, it calls a method named *reach_genesis_block* (inside the *Node* class) with the encountered block as an argument. This method checks whether by following the parent pointers starting from the given block, the genesis block can be reached or not. The node has detected a missing block, if the block tree does not contain a block before reaching the genesis block.

3.4 Node States

The waiting times of quick, medium and slow nodes until they create a block are crucial to the performance of piChain. Quick nodes are very eager to create blocks, while slow nodes are more patient. More specifically, as seen in section 2.3, a quick node creates a block immediately if it sees a new transaction, the medium nodes wait $(1+\epsilon)$ round-trip times (RTT's) and the slow nodes wait $(2+\epsilon)$ RTT's plus a random duration r. If r is chosen uniform at random in the interval $[0, n-1]$, where n is the number of nodes, there is a good chance that only a single slow node creates a block.[6]
The waiting time of a node is returned by the *get_patience* method in the *Node* class. The first time a slow node invokes this method, the random duration r is computed and remains fixed. After calculating the waiting time of a node, *get_patience* adds the accumulation time to get the final return value.
The interesting part is the RTT computation. A node basically keeps estimating the round-trip time to each of its peers and then takes the maximum among all estimates to compute the patience of a node. The node acts on the assumption that the delay to each of its peers is as long as the longest delay to any peer, such that the node does not create a block too early. To estimate the round-trip time to each peer, a ping-pong scheme is used. A node sends ping messages in regular intervals to each peer. A ping message just contains a timestamp that is taken at the moment the message is sent. If a node receives a ping message, it copies the timestamp and sends it back in form of a pong message. Once a node receives back a pong message, it can compute the difference of the current time and the received timestamp to get the round-trip time. Round-trip times are maintained in a dictionary that maps from a node ID to a RTT. Note that overestimations of the round-trip times do not affect the correctness of piChain but rather its performance.

[5]The number can be adjusted in the config module and depends on the application, namely on how long nodes are down on average and the expected RPS (Requests per second) rate.
[6]This can be shown using the Chernoff bound.

3.5 Strong Consistency

This subsection is about commits. The paragraphs are ordered in the chronological order of a commit and elaborate general challenges concerning the implementation.

After a node either has created a block and thus has been promoted to quick or if it already is quick, it tries to commit this block. If a commit is already running, it retries to commit the block after a certain timeout in hope that the currently running commit will then be over. The duration of the timeout is set to the expected time a node needs to commit a block, which can be computed based on the round-trip time estimates. If during the timeout a new transaction is received, the node creates a new block. Now the parent block does not have to be committed anymore since committing the new block implicitly commits all its ancestors. This can be implemented with a variable (*current_committable_block*) that always contains the most recent block to be committed.

The Paxos algorithm itself can basically be implemented one-to-one as described in Figure 2.3. There are some challenges however:

- A node not receiving a majority of try or propose acknowledgements:
  This happens if the connection to a majority of nodes is lost, for example because of a partition or node crashes. The node can deal with such a situation by deferring a method call once it begins a Paxos instance. The method call is invoked as soon as the expected time a node needs to commit a block is over. It then checks if the commit is still running. If yes, it is terminated to allow a new commit to happen and to not be blocked because of an already running commit.

- Handling of outdated Paxos messages:
  Outdated Paxos messages are messages that are received after a majority of acknowledgements has already been received and thus are not relevant anymore. One possibility to detect such messages involves sequence numbers: A node that wants to commit a block includes a unique request sequence number (*request_seq*) in its try and propose message. The peers receiving the try and propose message copy and include the request sequence number in their acknowledgement message. The node detects an outdated message if the request sequence number of an acknowledgement does not match the number in the corresponding request message (that is a try or propose message). Request sequence numbers are increased before sending a try message i.e., once Paxos is instantiated, after a majority of try-acknowledgements are received and after a majority of propose-acknowledgements are received.

- A node trying to commit a block that is not a descendant of the last committed block:
  This scenario might happen because of a partition. Assume a partition divides the network into a majority and a minority side. Both sides eventually have a quick node which continues to create blocks, but only the majority side can commit its blocks. As described earlier, such a situation leads to a fork of the blockchain. Once the partition is resolved, the quick node of the minority side might create a block and might try to commit this block, which is clearly not a descendant of the last committed block of the majority side. This scenario is illustrated in Figure 3.6. A simple solution would be to just reject blocks that are not descendants of the last committed block. This can be done in the first Phase of the Paxos algorithm.

- A node acting as a Paxos client and a Paxos server at the same time:
  Each node keeps a client state and a server state. In addition, during a broadcast all try, propose and commit messages are also "sent" to the sender itself.

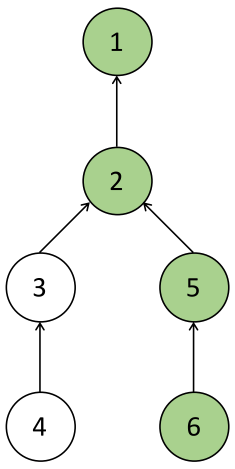

Figure 3.6: A blockchain fork caused by a partition of the network. The minority side creates block 3 and 4, while the majority side creates block 5 and 6. The majority side however can commit its blocks (committed blocks are colored in green). After the resolution of the partition, the quick node of the minority side may want to commit block 4, which is not a descendant of block 6, the last committed block.

3.5.1 Crash Recovery

This subsection is about the recovery of a node after a crash. If no data was written to disk, a node would lose everything, including its block tree.

LevelDB[7] is chosen as a storage library because it is simple to use and provides enough functionality for our purpose.[8] LevelDB is a fast key-value storage library written by Google. Keys and values are arbitrary byte arrays.

The following things have to be written to disk whenever they are changed and loaded from disk at initialization of a *Node* instance.

- The block tree: Whenever a block is added to the block tree (*nodes* dictionary in *Blocktree* class), it is stored to disk. The key is the encoded block ID and the value is the serialized block. If the blocks were not stored, the node might be lucky and might receive the missing blocks from another node. But this would be inefficient and if all nodes crashed, one after the other, before any missing block can be requested, all blocks would be lost. A reference to the head block (*head_block*) and to the last committed block (*committed_block*) is also stored.
- The counter: A global counter is used to create unique block and transaction IDs. The node has to store it, such that after a crash it knows where to proceed the enumeration.
- The Paxos server state variables: This includes b_{max}, b_{supp} and b_{prop}. They have to be stored, such that a Paxos server knows where he left off after a crash to still guarantee correctness during a Paxos instance.

3.6 Optimizations

This chapter elaborates on different strategies to optimize the algorithm regarding disk storage and performance.

3.6.1 Reduction of Blockchain Size

The disadvantage of a blockchain as a data structure is that it keeps growing over time. A monotonically growing data structure in the background of a consensus algorithm is not favorable. A possible solution is to adjust the genesis block of the blockchain over time and delete the ancestors of the new genesis block. But when is it safe to perform a genesis block change?

[7] `http://leveldb.org/`

[8] There exists a Python implementation of *LevelDB* called *Plyvel* (`https://github.com/wbolster/plyvel`).

A block that is committed on all the nodes can be the new genesis block for the following two reasons:

- Based on the strong consistency guarantee that Paxos provides for committed blocks, we know that a committed block is final and cannot be discarded even in the case of a blockchain fork.
- A node can only move to the new genesis block if it knows that all other nodes have committed the block. This is because a node that has not yet committed a block may be a node that crashed and thus does not know about the most recent blocks. If all other nodes then performed the genesis block change and deleted the ancestors, the crashed node would not be able to request the missing blocks anymore. Only if all nodes have committed a block, its ancestors can safely be deleted.

A way to implement this is to acknowledge a commit of a block to all other nodes and once a node received an acknowledgment from all other nodes that a certain block has been committed, it chooses this block to be the new genesis block. This is done with an *AckCommitMessage* (contained in the message module) that contains the block ID of the block that has been committed and which is broadcast to all other nodes. A node keeps a dictionary that maps from a block ID to an integer that counts how many times that block has been committed. Once a block has been committed by all nodes, it is set to the new genesis block and all its ancestors are deleted both from the block tree and from the disk. Note that since the genesis block is now adjusted over time, the *Blocktree* class needs an additional field that contains the current genesis block.

3.6.2 Quick Proposing

If a quick node receives a majority of propose acknowledgments in Phase 2, it may directly propose a new block in the next Paxos instance, i.e., it is allowed to skip Phase 1. This can be implemented with a boolean that tells the node if he is allowed to quick propose. It is set to true as soon as a majority of propose acknowledgments has been received. Since quick proposing grants a node the right to skip Phase 1, it is important to revoke it at any sign of deviation from the healthy state, specifically in the following scenarios:

- A node demotes to slow since we then know that the system was or is in an unhealthy state.
- A commit is terminated because the node did not receive enough acknowledgments. This may happen because of a partition and thus there might be another quick node.
- A node commits a block that was not created by the node itself, which means that there might be another quick node.

3.7 Networking

This chapter is about the networking module which includes establishing a peer-to-peer network and sending and receiving byte streams, respectively.

3.7.1 Twisted

Twisted [9] is an event-driven networking engine and is used for this project. Its core components are the following:

- **The Reactor**: The event loop of *Twisted* is part of the reactor. The reactor[9] waits for events and dispatches them to registered callback functions (also called request handlers) which then handle the events. As an example, events can be caused by an arriving network package or a method that was scheduled to be called after a delay.
- **Transports**: A transport represents the connection between two endpoints communicating over the network. This could for example be a TCP connection. Transports implement the *ITransport* interface that contains methods to write data to the connection and to get information about the peer of the connection.
- **Protocols**: Protocols describe how to process network events asynchronously. It consists of a Transport instance over which it receives data and can send data to the peer. A Protocol has to implement the *IProtocol* interface containing methods that define how to handle incoming data and what to do if a connection to a peer has been made or is lost.
- **Factories**: A new instance of a protocol class must be created for every connection. A factory is responsible for the creation and removal of those instances. It is basically the manager of all connections that are made to other peers. Information that must be persistent, i.e., kept across multiple connections, can be stored in the factory.

The above mentioned *Twisted* components are used in the following way:
The networking module consists of two classes, the *Connection* class and the *ConnectionManager* class. The *Connection* class is a subclass of *IntNStringReceiver* that is a predefined protocol of *Twisted*. It uses Length-Prefexing which prepends each message with its length. This allows to send and receive arbitrary strings of bytes. Each complete string that is received becomes a callback to the method *stringReceived*. The *ConnectionManager* class represents a factory. It creates *Connection* instances and keeps a consistent state among them. The

[9]It is called reactor because it reacts to things.

Connection class has a field called *connection_manager* which is set by the *ConnectionManager* to itself once it creates a *Connection* instance. This allows a *Connection* instance to have access to shared data between *Connection* instances and methods of the *ConnectionManager*. The *ConnectionManager* has an abstract receive-method for each message type (e.g., *receive_block*). Once a message is received, *stringReceived* of the *Connection* class is called which will delegate the call by invoking the corresponding receive-method of the *ConnectionManager* based on the message type[10]. The receive-methods however are overridden and implemented by the *Node* class, which is a subclass of the *ConnectionManager*, i.e., each *Node* in the system is basically a factory managing connections. This subclass relation allows a clean separation between the networking details and the logic of piChain. The relation between the *Node*, *Connection* and *ConnectionManager* class is illustrated in Figure 3.7.

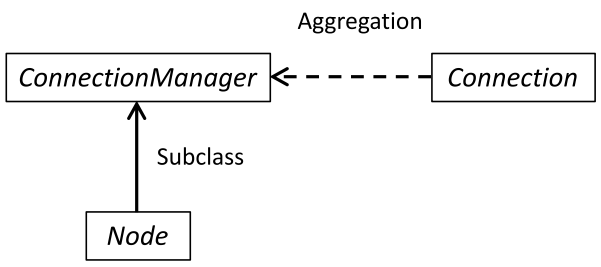

Figure 3.7: There is a subclass relation between *ConnectionManager* and *Node*. *ConnectionManager* and *Connection* is related through an aggregation i.e., each *Connection* instance has a reference to a *ConnectionManager* instance. For every piChain node there is a single *Node* instance and multiple *Connection* instances (one for each peer).

3.7.2 Peer-to-Peer Network

Establishing a peer-to-peer network is basically achieved by each node listening on a predefined port and then trying to connect to all other nodes.
Each node of the network is associated with a unique ID, a port and an IP address such that nodes can connect to each other: The *ConnectionManager* class (representing a node) contains a field called *peers*, which is a dictionary mapping from a node ID to an IP address and a port number. This dictionary is given as an argument to the *Node* constructor by the client using the piChain package.
There is a method *start_server* (in the *ConnectionManager*) that is part of the external interface and which can be called on a *Node* instance to start the whole networking process. It does three things:

[10]Each message is prepended with three bytes defining its type.

1. First it establishes a TCP server endpoint by listening on a TCP socket with a specific port. The port the node has to listen on is given in the peers dictionary associated with the ID of the node, which is also given as an argument to the *Node* constructor. Now other nodes can connect to this node.

2. Then the node tries to connect to other nodes by going over the *peers* dictionary and trying to connect to the given (IP, port) pairs. Each successful connection is represented by a *Connection* instance. To keep track of already connected peers, a dictionary called *peers_connection* stores all those *Connection* instances by mapping from the node ID of the peer to the *Connection* instance that represents the connection to the peer. The *Connection* class contains a method *connectionLost*, which is called whenever the connection to the peer of this connection is lost. If this happens, the node removes the peer from the *peers_connection* dictionary. Since some nodes may not be online at the time a node tries to connect to them, it has to attempt to connect in regular intervals until there is a successful connection to all the peers, which can be checked with the *peers_dictionary.* Also if a connection is lost, this looping call has to start again.

3. At last the reactor is started, which initiates the processes described above.

In a peer-to-peer network a node must be able to broadcast messages. The *broadcast* method in the *ConnectionManager* can be implemented by iterating over all the *Connection* instances in the *peers_connection* dictionary and calling *sendString()* on them.

Chapter 4

Evaluation

The implementation is evaluated based on unit and integration tests, on performance tests and on an application that is build on top of the piChain package.

4.1 Testing

4.1.1 Unit Tests

For unit testing the *unittest*[1] module in the Python standard library is used. Creating a test case is accomplished by subclassing *unittest.TestCase*. The individual tests can then be defined with methods whose names start with the letters "test". Each test can use the assert-methods provided by *unittest.TestCase* to verify certain conditions. The *setUp()* method is called before each test method and *tearDown()* is called after each test method. The following challenges were encountered while unit testing:

- One challenge is how to isolate units that interact with other units. For example if we want to test adding blocks to the block tree, we do not want to store blocks on the disk during the test, since testing the storage of blocks should be part of another unit test. We want to test units as isolated as possible to quickly identify the problem causing a bug.
  A possible solution to this problem is called mocking. Real objects or methods can be replaced with mock objects which mimic their behavior in a controlled way. The *unittest* module provides such mock objects.

- Testing the reception and broadcast of messages in the networking module would require to setup real connections. But real connections may breakdown, leading to nondeterministic results of the unit tests, which is unfavorable.
  A possible solution is to mock the connection, i.e., to not establish a

[1] https://docs.python.org/3/library/unittest.html

real connection in the unit test. *Twisted* provides fake transport implementations, for example the *twisted.test.proto_helpers.StringTransport* class which instead of sending bytes out over a network connection, writes them to a string which can then be inspected.

- Another challenge are timeouts. We want a unit test to provide an instantaneous result instead of waiting for timeouts to finish. This saves time because unit tests are executed many times to check that code changes did not break anything.
  Twisted provides a simple solution to this by faking the passage of time. The reactor can be replaced by an instance of the *twisted.internet.task.Clock* class, on which time can be advanced manually.

4.1.2 Integration Tests

The goal of integration tests is to test the system as a whole, i.e., to setup multiple nodes which communicate with each other over real connections. The major challenge is to fully automate those tests.
The idea to automate the integration tests is to use the Python *subprocess*[2] module to spawn multiple processes representing piChain nodes. After establishing the peer-to-peer network, the nodes run predefined scenarios. Relevant information like committed blocks is printed by nodes during execution such that it can be retrieved from the standard output pipe (*stdout*) after termination of the processes and can then be checked against test conditions. The standard error pipe (*stderr*) can be used to detect errors during execution.

The test scenarios can be divided into three categories:

- **Normal behavior**: These test scenarios test the behavior of the nodes while all nodes are online and connected to each other. To elaborately test the behavior of the nodes, all relevant states of the system, regarding the number of quick and medium nodes, are constructed.
- **Crash behavior**: These scenarios test what happens if certain nodes crash. As an example the quick node may crash while other nodes keep broadcasting transactions.
- **Partition behavior**: These scenarios test what happens if a partition occurs. A partition can be created by modifying the *broadcast* method of a node such that predefined nodes will not receive any messages from it anymore. In Python this can be done with run-time method patching, which modifies all instances of the modified class at run-time.

[2]https://docs.python.org/3/library/subprocess.html

4.2 Application: Distributed Database

This section is about the implementation of a distributed database on top of the piChain package. The architecture is illustrated in Figure 4.1.

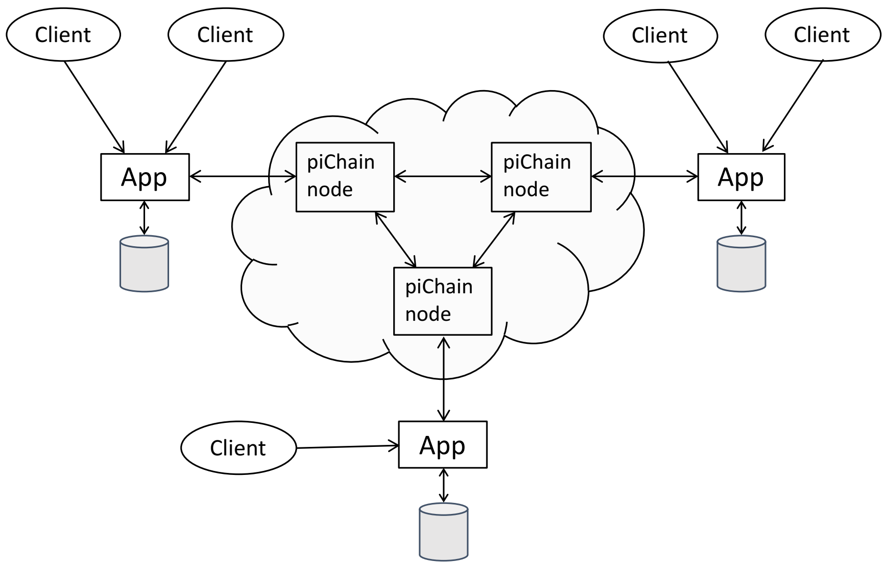

Figure 4.1: Each application instance has access to a piChain node and a local database. A client can connect to an arbitrary app and can send an executable database command. The app forwards the commands to the local piChain node. The communication between the piChain nodes is hidden from the applications. Once a command is committed, each piChain node informs the app instance it is associated with, which can then execute the command on the local database.

The implemented database is a key-value storage which supports three basic operations:

- *put(key, value)*: Store the given (key, value) pair in the database. If the key is already used, override the associated value with the new value.
- *get(key)*: Return the value associated with the given key. If the key does not exist, None is returned.
- *delete(key)*: Delete the (key, value) pair if the given key exists.

For the networking part, i.e., the communication with the client, *Twisted* is used. The app instance is represented by a *Twisted* factory which manages all the connections and keeps a consistent state among them. Each factory owns a *Node* instance and a *levelDB* instance representing the local database. At initialization, a factory starts listening on a predefined port such that clients can connect to it. Once a client is connected to a factory[3], it can send an executable database command. Read operations can be executed directly on the local database and the result can then be returned to the client, while write operations have to be committed first, such that their ordering is consistent among all database instances. A factory can send a command that has to be committed to its local *Node* instance by calling *make_txn()* with the command as an argument on it. Once the command has been committed, the *Node* instance invokes the *tx_committed* method with the committed commands. This method is stored in a field of the *Node* instance such that it can be implemented and set by the factory at creation of the *Node* instance. The factory can then execute the committed command on the local database.

4.3 Performance

This section is about performance tests of the piChain package and how it compares to *PySyncObj*, a popular Python implementation of Raft.
The performance plots were created as follows:
Each second a batch of transactions is sent to a running network of piChain nodes. The number of transactions in the batch corresponds to the requests per second (RPS) rate. The script sending the transactions increases the RPS rate in regular intervals until the nodes cannot commit the transactions anymore before a certain time has passed. With this information the maximum RPS rate can be computed.
To automate this for different cluster sizes, the Python *subprocess* module is used. For each node in the cluster and the script creating the transactions, a process is spawned. This is done for a predefined range of possible cluster sizes. The RPS limit computed for each cluster size can then be plotted.
Exactly the same performance test is implemented based on the *PySyncObj* package.

[3]Can be done with Telnet for example.

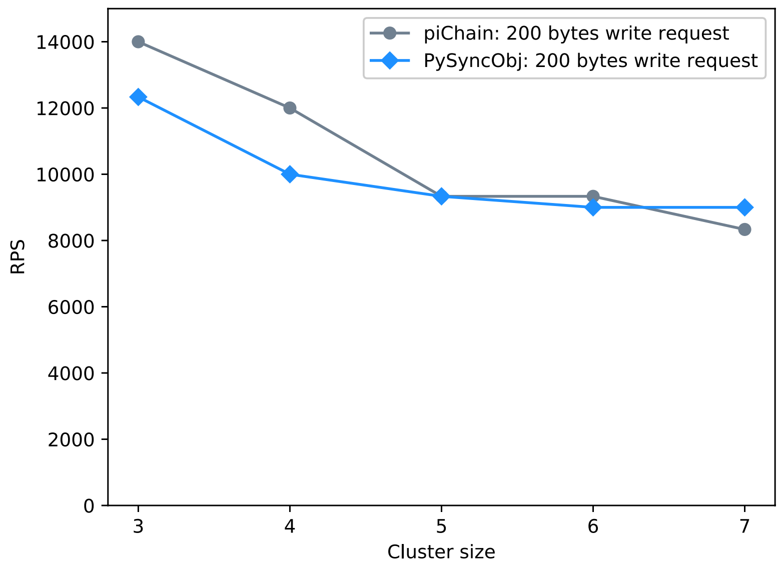

Figure 4.2: piChain and *PySyncObj* performance comparison. The plot shows the RPS limit of the piChain package and the *PySyncObj* package for different cluster sizes. The transaction size is kept constant at 200 bytes.

The plot in Figure 4.2 was created on a MacBook Pro with 16 GB DDR3 RAM at 1600 MHz and a 2.5 GHz Quad-Core Intel Core i7 processor. Each RPS value in the plot is an average over three independently computed RPS values. As one can see, both packages perform similarly and the RPS rate decreases with increasing cluster size because of a higher message load. Note that since all nodes run locally, there is almost no networking delay. The networking delay would in practice dominate the internal processing time of the algorithm, which would lead to plots that do not really measure the performance of the algorithm itself anymore.
To find out the bottleneck of the algorithm, profiling was done. The two main bottlenecks of the piChain package are serializing the data (35%) and writing it to disk (26.5%).

Chapter 5

Conclusion

In this thesis we implemented the proposed algorithm in the form of an easy-to-use library. We went through the relevant background theory of the algorithm and elaborated major challenges of the implementation. Also optimizations regarding storage usage and performance were made.
To show a real-world application of the library, a distributed fault-tolerant database was developed on top of the library.
Performance plots have shown that in a healthy state of the system the library can keep up with *PySyncObj*, a popular Python implementation of a state-of-the-art consensus algorithm called Raft.
In addition, piChain has the advantage of being a light protocol which has no heavy weight leader election subroutine and no heartbeat messages like in Raft.

5.1 Future Work

One might make the following improvements to the implementation:

- **Overlay network**: In the current peer-to-peer network, each node has an open connection to every other node. This might not be favorable if the number of nodes in the system is large, because the total number of connections increases quadratically with the number of nodes. A possible solution is to not connect every single pair of nodes, but rather relay messages until all nodes have received them. The network topology must make sure that any two nodes are able to exchange messages.

- **Byzantine fault tolerance**: A byzantine node is a node with arbitrary malicious behavior. piChain can be extended with byzantine tolerant elements, e.g., nodes could cryptographically sign their transactions in order to provide authenticity for them. Note that this most likely decreases the performance of the algorithm.

Bibliography

[1] Conrad Burchert and Roger Wattenhofer. piChain: When a Blockchain meets Paxos. *International Conference on Principles of Distributed Systems (OPODIS)*, 2017.

[2] Leslie Lamport. The Part-Time Parliament. *ACM Transactions on Computer Systems (TOCS)*, 1998.

[3] Leslie Lamport. Paxos Made Simple. *ACM Sigact News*, 2001.

[4] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System, 2008.

[5] Mike Burrows. The Chubby Lock Service for Loosely-Coupled Distributed Systems. *Symposium on Operating System Design and Implementation (OSDI)*, 2006.

[6] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. *USENIX Annual Technical Conference (USENIX ATC)*, 2014.

[7] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys (CSUR)*, 1990.

[8] Roger Wattenhofer. The Science of the Blockchain, 2016.

[9] Abe Fettig and Jessica McKellar. Twisted Network Programming Essentials: Event-driven Network Programming with Python, 2013.