The Cardano Consensus and Storage Layer

An IOHK technical report

Edsko de Vries
edsko@well-typed.com

Thomas Winant
thomas@well-typed.com

Duncan Coutts
duncan@well-typed.com
duncan.coutts@iohk.io

October 30, 2026

Contents

1 Introduction **7**

2 Overview **8**
2.1 Components . . . 8
2.1.1 Consensus protocols . . . 8
2.1.2 Ledger . . . 8
2.2 Design Goals . . . 9
2.2.1 Multiple consensus protocols . . . 9
2.2.2 Support for multiple ledgers . . . 9
2.2.3 Decouple consensus protocol from ledger . . . 9
2.2.4 Testability . . . 9
2.2.5 Adaptability and Maintainability . . . 10
2.2.6 Composability . . . 10
2.2.7 Predictable Performance . . . 11
2.2.8 Protection against DoS attacks . . . 11
2.3 History . . . 11

3 Non-functional requirements **12**
3.1 Network layer . . . 12
3.1.1 Header/Body Split (aka: Header submission) . . . 12
3.1.2 Block submission . . . 12
3.1.3 Transaction submission . . . 12
3.2 Security ”cost” concerns . . . 13
3.3 Hard time constraints . . . 13
3.4 Predictable resource requirements . . . 13
3.5 Resource registry . . . 13
3.5.1 Temporary registries . . . 14

I Consensus Layer **15**

4 Consensus Protocol **16**
4.1 Overview . . . 16
4.1.1 Chain selection . . . 16
4.1.2 The security parameter k . . . 17
4.2 The `ConsensusProtocol` Class . . . 18
4.2.1 Chain selection . . . 18
4.2.2 Ledger view . . . 19
4.2.3 Protocol state management . . . 19
4.2.4 Leader selection . . . 20
4.3 Connecting a block to a protocol . . . 21
4.4 Design decisions constraining the Ouroboros protocol family . . . 21

4.5 Permissive BFT . . . 22
4.5.1 Background . . . 22
4.5.2 Implementation . . . 22
4.5.3 Relation to the paper . . . 22
4.6 Praos . . . 24
4.6.1 Active slot coefficient . . . 24
4.6.2 Implementation . . . 24
4.6.3 Relation to the paper . . . 24
4.7 Combinator: Override the leader schedule . . . 24
4.8 Separation of responsibility between consensus and ledger . . . 26
4.8.1 Vision . . . 26
4.8.2 Practice . . . 27

5 Interface to the ledger **28**
5.1 Abstract interface . . . 28
5.1.1 Independent definitions . . . 28
5.1.2 Applying blocks . . . 29
5.1.3 Linking a block to its ledger . . . 29
5.1.4 Projecting out the ledger view . . . 30
5.2 Forecasting . . . 30
5.2.1 Introduction . . . 30
5.2.2 Code . . . 31
5.2.3 Ledger view . . . 31
5.3 Queries . . . 32
5.4 Abandoned approach: historical states . . . 32

6 Serialisation abstractions **33**
6.1 Serialising for storage . . . 33
6.1.1 Nested contents . . . 34
6.2 Serialising for network transmission . . . 36
6.2.1 Versioning . . . 37
6.2.2 CBOR-in-CBOR . . . 38
6.2.3 Serialised . . . 39
6.3 Annotations . . . 39
6.3.1 Slicing . . . 40

II Storage Layer **41**

7 Overview **42**
7.1 Components . . . 42
7.2 In memory . . . 42
7.2.1 Chain fragments . . . 42
7.2.2 Extended ledger state . . . 42

8 Immutable Database **43**
8.1 API . . . 44
8.1.1 Block Component . . . 45
8.1.2 Iterators . . . 45
8.2 Implementation . . . 46
8.2.1 Chunk layout . . . 47
8.2.2 Indices . . . 49

8.2.3 Recovery . . . 52

9 Volatile Database **55**
9.1 API . . . 56
9.2 Implementation . . . 57
9.2.1 Garbage collection . . . 59
9.2.2 Read-Append-Write lock . . . 59
9.2.3 Recovery . . . 60

10 Ledger Database **61**
10.1 In-memory representation . . . 61
10.2 On-disk . . . 63
10.2.1 Disk policy . . . 63
10.2.2 Initialisation . . . 63
10.3 Maintained by the Chain DB . . . 64

11 Chain Selection **65**
11.1 Comparing anchored fragments . . . 65
11.1.1 Introduction . . . 65
11.1.2 Precondition . . . 66
11.1.3 Definition . . . 66
11.2 Preliminaries . . . 67
11.2.1 Notation . . . 67
11.2.2 Properties . . . 68
11.3 Initialisation . . . 69
11.4 Adding new blocks . . . 69
11.4.1 Ignore . . . 69
11.4.2 Add to current chain . . . 70
11.4.3 Store, but don't change current chain . . . 70
11.4.4 Switch to a fork . . . 71
11.5 In-future check . . . 71
11.6 Sorting . . . 72

12 Chain Database **73**
12.1 Union of the Volatile DB and the Immutable DB . . . 73
12.1.1 Looking up blocks . . . 73
12.1.2 Iterators . . . 73
12.1.3 Followers . . . 74
12.2 Block processing queue . . . 75
12.3 Marking invalid blocks . . . 75
12.4 Effective maximum rollback . . . 76
12.5 Garbage collection . . . 76
12.5.1 GC delay . . . 77
12.6 Resources . . . 77

13 Mempool **78**
13.1 Consistency . . . 78
13.2 Caching . . . 79
13.3 TxSeq . . . 79
13.4 Capacity . . . 79

III Mini protocols **80**

14 Chain sync client **81**
14.1 Header validation . . . 81
14.2 Forecasting requirements . . . 81
14.3 Trimming . . . 81
14.4 Interface to the block fetch logic . . . 81

15 Mini protocol servers **82**
15.1 Local state query . . . 82
15.2 Chain sync . . . 82
15.3 Local transaction submission . . . 82
15.4 Block fetch . . . 82

IV Hard Fork Combinator **83**

16 Overview **84**
16.1 Introduction . . . 84

17 Time **85**
17.1 Introduction . . . 85
17.2 Slots, blocks and stability . . . 86
17.3 Definitions . . . 86
17.3.1 Time conversion . . . 86
17.3.2 Forecast range . . . 86
17.3.3 Safe zone . . . 87
17.4 Ledger restrictions . . . 87
17.4.1 Era transitions must be stable . . . 87
17.4.2 Size of the safezones . . . 88
17.4.3 Stability should not be approximated . . . 88
17.5 Properties . . . 89
17.5.1 Forecast ranges arising from safe zones . . . 89
17.5.2 Cross-fork conversions . . . 89
17.6 Avoiding time . . . 90
17.6.1 “Header in future” check . . . 90
17.6.2 Ahead-of-time “block in future” check . . . 91
17.6.3 “Immutable tip in future” check . . . 92
17.6.4 Scheduling actions for slot changes . . . 92
17.6.5 Switching on “deadline mode” in the network layer . . . 93

18 Misc stuff. To clean up. **94**
18.1 Ledger . . . 94
18.1.1 Invalid states . . . 94
18.2 Keeping track of time . . . 94
18.3 Failed attempts . . . 94
18.3.1 Forecasting . . . 94

V Testing **96**

19 Reaching consensus **97**
19.1 Dire-but-not-to-dire . . . 97

20 The storage layer **98**

VI Future Work **99**

21 Ouroboros Genesis **100**
21.1 Introduction . . . 100
21.1.1 Background: understanding the Longest Chain rule . . . 100
21.1.2 Nodes joining late . . . 101
21.1.3 The Density rule . . . 102
21.2 Properties of the Density rule . . . 102
21.2.1 Equivalence to the Longest Chain rule . . . 102
21.2.2 Honest versus adversarial blocks . . . 103
21.2.3 The original genesis rule . . . 104
21.3 Fragment selection . . . 104
21.4 Prefix selection . . . 107
21.4.1 Preferred prefix . . . 107
21.4.2 Algorithm . . . 107
21.4.3 Prefix selection on headers . . . 108
21.4.4 Known density . . . 110
21.4.5 Propagation delays . . . 110
21.5 Avoiding long rollbacks . . . 111
21.5.1 Representative sample . . . 111
21.5.2 Becoming alert . . . 112
21.5.3 Avoiding DoS attacks . . . 113
21.5.4 Falling behind . . . 113
21.5.5 Block production . . . 113
21.6 Implementation concerns . . . 113
21.6.1 Chain selection in the chain database . . . 113
21.6.2 Abstract chain selection interface . . . 114
21.6.3 Possible optimisations . . . 115

22 Dealing with extreme low-density chains **116**
22.1 Introduction . . . 116
22.2 Background . . . 117
22.2.1 Recap: ledger state, ledger view and forecasting . . . 117
22.2.2 Recap: stability windows . . . 118
22.2.3 Tension with chain selection . . . 119
22.2.4 Single-gap case . . . 120
22.3 Pre-genesis . . . 120
22.3.1 Damage mitigation . . . 120
22.3.2 Damage analysis . . . 122
22.4 Post-genesis . . . 123
22.4.1 Pre-disaster genesis window . . . 123
22.4.2 Post-disaster genesis window . . . 124
22.4.3 (No) need for gap jumping . . . 125
22.4.4 In the absence of adversaries . . . 125

23 Epoch Boundary Blocks **126**
23.1 Introduction . . . 126
23.2 Logical slot/block numbers . . . 127
23.3 Eliminating EBBs altogether . . . 127

24 Miscellaneous **128**
24.1 On abstraction . . . 128
24.2 On-disk ledger state . . . 128
24.3 Transaction TTL . . . 128
24.4 Block based versus slot based . . . 128
24.5 Eliminating safe zones . . . 128
24.6 Eliminating forecasting . . . 128
24.7 Open kinds . . . 129
24.8 Relax requirements on time conversions . . . 129
24.9 Configuration . . . 129
24.10 Specialised chain selection data structure . . . 129
24.11 Dealing with clock changes . . . 129

VII Conclusions **131**

25 Technical design decisions **132**
25.1 Classes versus records . . . 132
25.2 Top-level versus associated type families . . . 132

26 Conclusions **133**

VIII Appendices **134**

A Byron **135**
A.1 Update proposals . . . 135
A.1.1 Moment of hard fork . . . 135
A.1.2 The update mechanism for the `ProtocolVersion` . . . 135
A.1.3 Initiating the hard fork . . . 136
A.1.4 Software versions . . . 137

B Shelley **138**
B.1 Update proposals . . . 138
B.1.1 Moment of the hard fork . . . 138
B.1.2 The update mechanism for the protocol version . . . 138
B.1.3 Initiating the hard fork . . . 139
B.2 Forecasting . . . 139

Chapter 1

Introduction

The Cardano Consensus and Storage layer, or *the consensus layer* for short, is a critical piece of infrastructure in the Cardano Node. It orchestrates between the *network layer* below it and the *ledger layer* above it.

The network layer is a highly concurrent piece of software that deals with low-level concerns; its main responsibility is to transmit data efficiently across the network. Although it primarily transmits blocks and block headers, it does not interpret them and does not need to know much about them. In the few cases where it *does* need to make some block-specific decisions, it calls back into the consensus layer to do so.

The ledger layer by contrast exclusively deals with high-level concerns. It is entirely stateless: its main responsibility is to define a single pure function describing how the ledger state is transformed by blocks (verifying that blocks are valid in the process). It is only concerned with linear history; it is not aware of the presence of multiple competing chains or the roll backs required when switching from one chain to another. We do require that the ledger layer provides limited *lookahead*, computing (views on near) *future* ledger states (required to be able to validate block headers without access to the corresponding block bodies)

The consensus layer mediates between these two layers. It includes a bespoke storage layer that provides efficient access to the current ledger state as well as recent *past* ledger states (required in order to be able to validate and switch to competing chains). The storage layer also provides direct access to the blocks on the blockchain itself, so that they can be efficiently streamed to clients (via the network layer). When there are competing chains, the consensus layer decides which chain is preferable and should be adopted, and it decides when to *contribute* to the chain (produce new blocks). All "executive decisions" about the chain are made in and by the consensus layer.

Lastly, as well we see, the consensus layer is highly abstract and places a strong emphasis on compositionality, making it usable with many different consensus algorithms and ledgers. Importantly, compositionality enables the *hard fork combinator* to combine multiple ledgers and regard them as a single blockchain.

The goal of this document is to outline the design goals for the consensus layer, how we achieved them, and where there is still scope for improvement. We will both describe *what* the consensus layer is, and *why* it is the way it is. Throughout we will also discuss what *didn't* work, approaches we considered but rejected, or indeed adopted but later abandoned; discussing these dead ends is sometimes at least as informative as discussing the solution that did work.

We will consider some of the trade-offs we have had to make, how they affected the development, and discuss which of these trade-offs should perhaps be reconsidered. We will also take a look at how the design can scale to facilitate future requirements, and which requirements will be more problematic and require more large-scale refactoring.

The target audience for this document is primarily developers working on the consensus layer. It may also be of more broader interest to people generally interested in the Cardano blockchain, although we will assume that the reader has a technical background.

Chapter 2

Overview

2.1 Components

2.1.1 Consensus protocols

The consensus protocol has two primary responsibilities:

Chain selection Competing chains arise when two or more nodes extend the chain with different blocks. This can happen when nodes are not aware of each other's blocks due to temporarily network delays or partitioning, but depending on the particular choice of consensus algorithm it can also happen in the normal course of events. When it happens, it is the responsibility of the consensus protocol to choose between these competing chains.

Leadership check In proof-of-work blockchains any node can produce a block at any time, provided that they have sufficient hashing power. By contrast, in proof-of-stake time is divided into *slots*, and each slot has a number of designated *slot leaders* who can produce blocks in that slot. It is the responsibility of the consensus protocol to decide on this mapping from slots to slot leaders.

The consensus protocol will also need to maintain its own state; we will discuss state management in more detail in section 7.2.

2.1.2 Ledger

The role of the ledger is to define what is stored *on* the blockchain. From the perspective of the consensus layer, the ledger has three primary responsibilities:

Applying blocks The most obvious and most important responsibility of the ledger is to define how the ledger state changes in response to new blocks, validating blocks at it goes and rejecting invalid blocks.

Applying transactions Similar to applying blocks, the ledger layer also must provide an interface for applying a single transaction to the ledger state. This is important, because the consensus layer does not just deal with previously constructed blocks, but also constructs *new* blocks.

Ticking time Some parts of the ledger state change due to the passage of time only. For example, blocks might *schedule* some changes to be applied later, and then when the relevant slot arrives those changes should be applied, independent from any blocks.

Forecasting Some consensus protocols require limited information from the ledger. In Praos, for example, a node's probability of being a slot leader is proportional to its stake, but the stake distribution is something that the ledger keeps track of. We refer to this as a *view* on the ledger, and we require not just that the ledger can give us a view on the *current* ledger state, but also *predict* what that

view will be for slots in the near future. We will discuss the motivation for this requirement in section 3.1.1.

The primary reason for separating out "ticking" from applying blocks is that the consensus layer is responsible to the leadership check (section 2.1.1), and when we need to decide if we should be producing a block in a particular slot, we need to know the ledger state at that slot (even though we don't have a block for that slot *yet*). It is also required in the mempool; see chapter 13.

2.2 Design Goals

2.2.1 Multiple consensus protocols

From the beginning it was clear that we would need support for multiple consensus algorithms: the Byron era uses a consensus algorithm called (Permissive) BFT (section 4.5) and the Shelley era uses a consensus algorithm called Praos (section 4.6). Moreover, the Cardano blockchain is a *hybrid* chain where the prefix of the chain runs Byron (and thus uses BFT), and then continues with Shelley (and thus uses Praos); we will come back to the topic of composing protocols when we discuss the hard fork combinator (chapter 16). It is therefore important that the consensus layer abstracts over a choice of consensus protocol.

2.2.2 Support for multiple ledgers

For much the same reason that we must support multiple consensus protocols, we also have to support multiple ledgers. Indeed, we expect more changes in ledger than in consensus protocol; currently the Cardano blockchain starts with a Byron ledger and then transitions to a Shelley ledger, but further changes to the ledger have already been planned (some intermediate ledgers currently code-named Allegra and Mary, as well as larger updates to Goguen, Basho and Voltaire). All of the ledgers (Shelley up to including Voltaire) use the Praos consensus algorithm (potentially extended with the genesis chain selection rule, see chapter 21).

2.2.3 Decouple consensus protocol from ledger

As we saw above (section 2.2.2), we have multiple ledgers that all use the same consensus protocol. We therefore should be able to define the consensus protocol *independent* from a particular choice of ledger, merely defining what the consensus protocol expects from the ledger (we will see what this interface looks like in chapter 5).

2.2.4 Testability

The consensus layer is a critical component of the Cardano Node, the software that runs the Cardano blockchain. Since the blockchain is used to run the ada cryptocurrency, it is of the utmost importance that this node is reliable; network downtime or, worse, corruption of the blockchain, cannot be tolerated. As such the consensus layer is subject to much stricter correctness criteria than most software, and must be tested thoroughly. To make this possible, we have to design for testability.

- We must be able to simulate various kinds of failures (disk failures, network failures, etc.) and verify that the system can recover.
- We must be able to run *lots* of tests which means that tests need to be cheap. This in turn will require for example the possibility to swap the cryptographic algorithms for much faster "mock" crypto algorithms.
- We must be able to test how the system behaves under certain expected-but-rare circumstances. For example, under the Praos consensus protocol it can happen that a particular slot has multiple leaders.

We should be able to test what happens when this happens repeatedly, but the leader selection is a probabilistic process; it would be difficult to set up test scenarios to test for this specifically, and even more difficult to set things up so that those scenarios are *shrinkable* (leading to minimal test cases). We must therefore be able to "override" the behaviour of the consensus protocol (or indeed the ledger) at specific points.

- We must be able to test components individually (rather than just the system as a whole), so that if a test fails, it is much easier to see where something went wrong.

2.2.5 Adaptability and Maintainability

The Cardano Node began its life as an ambitious replacement of the initial implementation of the Cardano blockchain, which had been developed by Serokell. At the time, the Shelley ledger was no more than a on-paper design, and the Praos consensus protocol existed only as a research paper. Moreover, since the redesign would be unable to reuse any parts of the initial implementation, even the Byron ledger did not yet exist when the consensus layer was started. It was therefore important from the get-go that the consensus layer was not written for a specific ledger, but rather abstract over a choice of ledger and define precisely what the responsibilities of that ledger were.

This abstraction over both the consensus algorithm and the ledger is important for other reasons, too. As we've mentioned, although initially developed to support the Byron ledger and the (Permissive) BFT consensus algorithm, the goal was to move to Shelley/Praos as quickly as possible. Moreover, additional ledgers had already been planned (Goguen, Basho and Voltaire), and research on consensus protocols was (and still is) ongoing. It was therefore important that the consensus layer could easily be adapted.

Admittedly, adaptability does not *necessarily* require abstraction. We could have built the consensus layer against the Byron ledger initially (although we might have had to wait for it to be partially completed at least), and then generalise as we went. There are however a number of downsides to this approach.

- When working with a concrete interface, it is difficult to avoid certain assumptions creeping in that may hold for this ledger but will not hold for other ledgers necessarily. When such assumptions go unnoticed, it can be costly to adjust later. (For one example of such an assumption that nonetheless *did* go unnoticed, despite best efforts, and took a lot of work to resolve, see chapter 17 on removing the assumption that we can always convert between wallclock time and slot number.)

- IOHK is involved in the development of blockchains other than the public Cardano instance, and from the start of the project, the hope was that the consensus layer can be used in those projects as well. Indeed, it is currently being integrated into various other IOHK projects.

- Perhaps most importantly, if the consensus layer only supports a single, concrete ledger, it would be impossible to *test* the consensus layer with any ledgers other than that concrete ledger. But this means that all consensus tests need to deal with all the complexities of the real ledger. By contrast, by staying abstract, we can run a lot of consensus tests with mock ledgers that are easier to set up, easier to reason about, more easily instrumented and more amenable to artificially creating rare circumstances (see section 2.2.4).

Of course, abstraction is also just good engineering practice. Programming against an abstract interface means we are clear about our assumptions, decreases dependence between components, and makes it easier to understand and work with individual components without having to necessarily understand the entire system as a whole.

2.2.6 Composability

The consensus layer is a complex piece of software; at the time we are writing this technical report, it consists of roughly 100,000 lines of code. It is therefore important that we split it into into small

components that can be understood and modified independently from the rest of the system. Abstraction, discussed in section 2.2.5, is one technique to do that, but by no means the only. One other technique that we make heavy use of is composability. We will list two examples here:

- As discussed in section 2.2.1 and section 2.2.2, the Cardano blockchain has a prefix that runs the BFT consensus protocol and the Byron ledger, and then continues with the Praos consensus protocol and the Shelley ledger. We do not however define a consensus protocol that is the combination of Byron and Praos, nor a ledger that is the combination of Byron and Shelley. Instead, the *hard fork combinator* (chapter 16) makes it possible to *compose* consensus protocols and ledgers: construct the hybrid consensus protocol from an implementation of BFT and an implementation of Praos, and similarly for the ledger.
- We mentioned in section 2.2.4 that it is important that we can test the behaviour of the consensus layer under rare-but-possible circumstances, and that it is therefore important that we can override the behaviour of the consensus algorithm in tests. We do not accomplish this however by adding special hooks to the Praos consensus algorithm (or any other); instead we define another combinator that takes the implementation of a consensus algorithm and *adds* additional hooks for the sake of the testing infrastructure. This means that the implementation of Praos does not have to be aware of testing constraints, and the combinator that adds these testing hooks does not need to be aware of the details of how Praos is implemented.

2.2.7 Predictable Performance

Make sure node operators do not set up nodes for ”normal circumstances” only for the network to go down when something infrequent (but expected) occurs. (This is not about malicious behaviour, that’s the next section).

Duncan suitable section.

2.2.8 Protection against DoS attacks

Brief introduction to asymptotic attacker/defender costs. (This is just an overview, we come back to these topics in more detail later.)

Duncan suitable section.

2.3 History

Duncan suitable section.

- Briefly describe the old system (`cardano-sl`) the decision to rewrite it
- Briefly discuss the OBFT hard fork.

Chapter 3

Non-functional requirements

This whole chapter is Duncan-suitable :)

Duncan suitable section.

3.1 Network layer

This report is not intended as a comprehensive discussion of the network layer; see [4] instead. However, in order to understand some of the design decisions in the consensus layer we need to understand some of the requirements imposed on it by the network layer.

TODOs:

- Highlight relevant aspects of the design of the network layer
- Discuss requirements this imposes on the consensus layer Primary example: Forecasting.
- How do we keep the overlap between network and consensus as small as possible? Network protocols do not involve consensus protocols (chain sync client is not dependent on chain selection). Chain sync client + ”pre chain selection” + block download logic keeps things isolated.
- Why do we even want to validate headers ahead of time? (Thread model etc.) (Section for Duncan?). Section with a sketch on an analysis of the amortised cost for attackers versus our own costs to defend against it (”budget for work” that grows and shrinks as you interact with a node).

3.1.1 Header/Body Split (aka: Header submission)

Discuss the chain fragments that we store per upstream node. Discuss why we want to validate headers here – without a full ledger state (necessarily so, since no block bodies – can’t update ledger state): to prevent DoS attacks. (section 5.2 contains a discussion of this from the point of view of the ledger). Forward reference to the chain sync client (chapter 14). Discuss why it’s useful if the chain sync client can race ahead for *performance* (why it’s required for chain selection is the discussed in section 5.2.3).

See also section on avoiding the stability window (section 22.3).

3.1.2 Block submission

Forward reference to section 15.4.

3.1.3 Transaction submission

Mention that these are defined entirely network side, no consensus involvement (just an abstraction over the mempool).

3.2 Security ”cost” concerns

TODO: Look through the code and git history to find instances of where we one way but not the other because it would give an attacker an easy way to make it do lots of work (where were many such instances).

Fragile. Future work: how might be make this less brittle? Or indeed, how might we test this?

Counter-examples (things we don't want to do)

- Parallel validation of an entire epoch of data (say, crypto only). You might do a lot of work before realising that that work was not needed because of an invalid block in the middle.

Future work: opportunities for parallelism that we don't yet exploit (important example: script evaluation in Goguen).

3.3 Hard time constraints

Must produce a block on time, get it to the next slot leader

Bad counter-example: reward calculation in the Shelley ledger bad (give examples of why).

3.4 Predictable resource requirements

make best == worst

(not *just* a security concern: a concern even if every node honest)

3.5 Resource registry

In order to deal with resource allocation and deallocation, we use the abstraction of the `ResourceRegistry`. The resource registry is a generalization of the **`bracket`** pattern. Using a bracket imposes strict rules on the scope of the allocated variables, as once the body of the **`bracket`** call finishes, the resources will be deallocated.

On some situations this is too constraining and resources, despite the need to ensure they are deallocated, need to live for longer periods or be shared by multiple consumers. The registry itself lives in a **`bracket`**-like environment, but the resources allocated in it can be opened and closed at any point in the lifetime of the registry and will ultimately be closed when the registry is being closed if they are still open at that time.

The allocation function for a given resource will run with exceptions unconditionally masked, therefore in an uninterruptible fashion. Because of this, it is important that such functions are fast. Note that once the allocation call finishes, the resource is now managed by the registry and will be deallocated in case an exception is thrown.

A resource that is allocated will be returned coupled with its `ResourceKey` that can be used to release the resource as it holds a reference to the registry in which it is allocated.

Special care must be used when resources are dependent on one another. For example, a thread allocated in the registry might hold some mutable reference to a file handle that is replaced at certain points during the execution. In such cases, the sequence of deallocation must take this into account and deallocate the resources in reverse order of dependency.

Also when deallocating the resources, we must ensure that the right order of deallocation is preserved. Right order here means that as resources that were allocated later than others could potentially use the latter, the later ones should probably be deallocated before the earlier ones (unless otherwise taken care of). Resources in the registry are indexed by their *age* which is a meaningless backwards counter. A resource is considered older than another if its age is greater than the one of the other resource. Conversely, a resource is considered younger if the opposite holds.

3.5.1 Temporary registries

When some resources are not meant to be directly allocated in a registry, one can take advantage of temporary resource registries as a temporary container for those resources. For this purpose, the `WithTempRegistry` is made available. It is basically a `StateT` computation which will check that the resource was indeed transferred properly to the returned value and then the registry will vanish.

Using this construction will ensure that if an exception is thrown while the resources are being allocated but before they are kept track by the resulting state, they will be released as the registry itself will close in presence of an exception. Note that once the resource was transferred to the final state, no more tracking is performed and the resource could be leaked. It is then responsibility of the resulting state to eventually deallocate the resource, and so that resulting state must itself ultimately be for example tracked by a registry.

Temporary registries are useful when we want to run localized allocations and checks, i.e. allocations and checks that use implementation details that should remain hidden for upper layers. This is specially useful when we expect to run a computation that will provide a result that holds some API which references some internal data types which are not directly accessible from the API, but we still want to run some checks on those internal data types. [1] The resulting value will be transitively closable from the general registry through functions on the API (deallocating its resources), but the actual value is not accessible to perform the checks usually involved with `WithTempRegistry`.

Note that if we just run the inner computation and return the API value, there is a short time span during which an exception would leak the resources as the API that can close the resources is not yet included in any exception handler that will close it. A special case of this situation is when we run a computation with a (*upper*) registry and we want to perform a localized allocation and checks on an internal component. The combinator `runInnerWithTempRegistry` is provided. This combinator will make sure that the composite resource is allocated in the *upper* registry before closing the inner one (therefore performing the allocation checks against the resulting inner state), and thus in presence of an exception the resources will be safely deallocated. There are a couple of subtleties here that are worth being mentioned:

- There is a short time span during which the inner registry has not yet vanished, but the resource has already been allocated in the greater registry. An exception in this exact moment will lead to double freeing a resource.
- Unless the greater resulting state has some way of accessing the returned inner state, the function that performs the checks will necessarily be degenerate (i.e. `const (const True)`). If we had a way to access the inner returned state, we could run the checks, but adding this implies leaking the inner state, its representation and functions, and most of the time this is not desirable as those are usually private implementation details.

Alternative: Specialization to `TrackingTempRegistry` The `TempRegistry` concept could be specialized in a narrower concept, a `TrackingTempRegistry` which would essentially be equivalent to a `TempRegistry () m` and no checks would be preformed on the resulting state. This way some redundant situations could be simplified. Note that this is also equivalent to using a special `ResourceRegistry` which works as a normal `ResourceRegistry` but instead of deallocating them, it would just untrack all the allocated resources when going out of scope.

[1] See `VolatileDB.openDB` for an example. The inner computation allocates and performs checks against a `OpenState blk h` whereas the returned value is a `VolatileDB` which has a `close` function but otherwise has no direct access to the internal `OpenState blk h`.

Part I

Consensus Layer

Chapter 4

Consensus Protocol

4.1 Overview

4.1.1 Chain selection

Chain selection is the process of choosing between multiple competing chains, and is one of the most important responsibilities of a consensus protocol. When choosing between two chains, in theory any part of those chains could be relevant; indeed, the research literature typically describes chain selection as a comparison of two entire chains (sections 4.5.3 and 4.6.3). In practice that is not realistic: the node has to do chain selection frequently, and scanning millions of blocks each time to make the comparison is of course out of the question.

The consensus layer keeps the most recent headers as a *chain fragment* in memory (section 7.2); the rest of the chain is stored on disk. Similarly, we keep a chain fragment of headers in memory for every (upstream) node whose chain we are following and whose blocks we may wish to adopt (chapter 14). Before the introduction of the hard fork combinator chain selection used to be given these fragments to compare; as we will discuss in section 16.1, however, this does not scale so well to hybrid chains.

It turns out, however, that it suffices to look only at the headers at the very tip of the chain, at least for the class of consensus algorithms we need to support. The exact information we need about that tip varies from one protocol to the other, but at least for the Ouroboros family of consensus protocols the essence is always the same: we prefer longer chains over shorter ones (justifying *why* this is the right choice is the domain of cryptographic research and well outside the scope of this report). In the simplest case, the length of the chain is *all* that matters, and hence the only thing we need to know about the blocks at the tips of the chains is their block numbers.[2]

This does beg the question of how to compare two chains when one (or both) of them are empty, since now we have no header to compare. We will resolve this by stating the following fundamental assumption about *all* chain selection algorithms supported by the consensus layer:

Assumption 4.1 (Prefer extension)**.** The extension of a chain is always preferred over that chain.

A direct consequence of assumption 4.1 is that a non-empty chain is always preferred over an empty one,[3] but we will actually need something stronger than that: we insist that shorter chains can never be preferred over longer ones:

Assumption 4.2 (Never Shrink)**.** A shorter chain is never preferred over a longer chain.

[2] It doesn't actually matter if the actual block headers contain a block number or not; if they don't, we can add a "virtual field" to the in-memory representation of the block header. For block headers that *do* include a block number (which is the case for the Cardano chain), header validation verifies that the block number is increasing. Note that EBBs complicate this particular somewhat; see page 127.

[3] Comparing empty chain *fragments*, introduced in section 7.2.1, is significantly more subtle, and will be discussed in section 11.1.

Assumption 4.2 does not say anything about chains of equal length; this will be important for Praos (section 4.6). An important side-note here is that the Ouroboros Genesis consensus protocol includes a chain selection rule (the genesis rule) that violates assumption 4.2 (though not assumption 4.1); it also cannot be defined by only looking at the tips of chains. It will therefore require special treatment; we will come back to this in chapter 21.

4.1.2 The security parameter k

When the Cardano blockchain was first launched, it was using a consensus protocol that we now refer to as Ouroboros Classic [10]. The re-implementation of the consensus layer never had support for Ouroboros Classic, instead using Ouroboros BFT [9] as a transitional protocol towards Ouroboros Praos [6], which is the consensus protocol in use at the time of writing, with plans to switch to Ouroboros Genesis [1] relatively soon (chapter 21).

Both Ouroboros Classic and Ouroboros Praos are based on a chain selection rule that imposes a maximum rollback condition: alternative chains to a node's current chain that fork off more than a certain number of blocks ago are never considered for adoption. This limit is known as the *security parameter*, and is usually denoted by k; at present $k = 2160$ blocks. The Ouroboros analysis shows that consensus will be reached despite this maximum rollback limitation; indeed, this maximum rollback is *required* in order to reach consensus (we discuss this in some detail in section 21.1.1).

For Ouroboros BFT and Ouroboros Genesis the situation is slightly different:

- Ouroboros BFT does not impose a maximum rollback, but adding such a requirement does not change the protocol in any fundamental way: the analysis for Ouroboros Praos shows that nodes will not diverge more than k blocks, and since BFT converges much quicker than that, adding this (large) maximum rollback requirement does not change anything.

- The analysis that shows that nodes will not diverge by more than k blocks does of course not apply to new nodes joining the system. Indeed, when using Ouroboros Praos, such nodes are vulnerable to an attack where an adversary with some stake (does not have to be much) presents the newly joining node with a chain that diverges by more than k blocks from the honest chain, at which point point the node would become unable to switch to the real chain. Solving this is the purview of Ouroboros Genesis.

  Like Ouroboros BFT, Ouroboros Genesis likewise does not impose a maximum rollback, *but* the analysis [1] shows that when nodes are up to date, they can employ the Ouroboros Praos rule (i.e., the rule *with* the maximum rollback requirement). This is not true when the node is behind and is catching up, but the main goal of chapter 21 is to show how we can nonetheless avoid rollbacks exceeding k blocks even when a node is catching up.

Within the consensus layer we therefore assume that we *always* have a limit k on the number of blocks we might have to rollback. We take advantage of this in many ways; here we just mention a few:

- We use it as an organising principle in the storage layer (chapter 7), dividing the chain into a part that we know is stable (the "immutable chain"), and a part near the tip that is still subject to rollback (the "volatile chain"). Block lookup into the immutable chain is very efficient, and since the vast majority of the chain is immutable, this helps improve overall efficiency of the system.

- When we switch to a new fork by rolling back and then adopting some new blocks, those new blocks must be verified against the ledger state as it was at the point we rolled back to. This means we must be able to construct historical ledger states. In principle this is always possible, as we can always replay the entire chain, but doing so would be expensive. However, since we have a limit on the maximum rollback, we also have a limit on how old the oldest ledger state is we might have to reconstruct; we take advantage of this in the Ledger Database (chapter 10) which can efficiently reconstruct any of those k historical ledger states.

- We need to keep track of the chains of our peer nodes in order to be able to decide whether or not we might wish to switch to those chains (chapter 14). For consensus protocols based on a longest chain rule (such as Ouroboros Praos), this means that we would need to download and verify enough blocks from those alternative chains that the alternative chain becomes longer than our own. Without a maximum rollback, this would be an unbounded amount of work as well as an unbounded amount of data we would have to store. A maximum rollback of k, however, means that validating (and storing) $k + 1$ blocks should be sufficient.[4]

Of course, a maximum rollback may be problematic in the case of severe network outages that partition the nodes for extended periods of time (in the order of days). When this happens, the chains will diverge and recovering converge will need manual intervention; this is true for any of the consensus protocols mentioned above. This manual intervention is outside the scope of this report.

4.2 The `ConsensusProtocol` Class

We model consensus protocols as a single class called `ConsensusProtocol`; this class can be considered to be the central class within the consensus layer.

```
class (..) => ConsensusProtocol p where
```

The type variable p is a type-level tag describing a particular consensus protocol; if Haskell had open kinds[5], we could say `(p :: ConsensusProtocol)`. All functions within this class take an argument of type

```
data family ConsensusConfig p :: Type
```

This allows the protocol to depend on some static configuration data; what configuration data is required will vary from protocol to protocol.[6] The rest of the consensus layer does not really do much with this configuration, except make it available where required; however, we do require that whatever the configuration is, we can extract k from it:

```
protocolSecurityParam :: ConsensusConfig p -> SecurityParam
```

For example, this is used by the chain database to determine when blocks can be moved from the volatile DB to the immutable DB (section 7.1). In the rest of this section we will consider the various parts of the `ConsensusProtocol` class one by one.

4.2.1 Chain selection

As mentioned in section 4.1.1, chain selection will only look at the headers at the tip of the ledger. Since we are defining consensus protocols independent from a concrete choice of ledger, however (section 2.2.3), we cannot use a concrete block or header type. Instead, we merely say that the chain selection requires *some* view on headers that it needs to make its decisions:

```
type family SelectView p :: Type
type SelectView p = BlockNo
```

The default is `BlockNo` because as we have seen this is all that is required for the most important chain selection rule, simply preferring longer chains over shorter ones. It is the responsibility of the glue code that connects a specific choice of ledger to a consensus protocol to define the projection from a concrete block type to this `SelectView` (4.3). We then require that these views must be comparable

```
class (Ord (SelectView p), ..) => ConsensusProtocol p where
```

[4] For chain selection algorithms such as Ouroboros Genesis which are based on properties of the chains near their *intersection point* rather than near their tips this is less relevant.

[5] We will come back to this in section 24.7.

[6] Explicitly modelling such a required context could be avoided if we used explicit records instead of type classes; we will discuss this point in more detail in section 25.1.

and say that one chain is (strictly) preferred over another if its `SelectView` is greater. If two chains terminate in headers with the *same* view, neither chain is preferred over the other, and we could pick either one (we say they are equally preferable).

Later in this chapter we will discuss in detail how our treatment of consensus algorithms differs from the research literature (sections 4.5 and 4.6), and in chapter 11 we will see how the details of how chain selection is implemented in the chain database; it is worth pointing out here, however, that the comparison based on `SelectView` is not intended to capture

- chain validity
- the intersection point (checking that the intersection point is not too far back, preserving the invariant that we never roll back more than k blocks, see section 4.1.2)

Both of these responsibilities would require more than seeing just the tip of the chains. They are handled independent of the choice of consensus protocol by the chain database, as discussed in chapter 11.

When two *candidate* chains (that is, two chains that aren't our current) are equally preferable, we are free to choose either one. However, when a candidate chain is equally preferable to our current, we *must* stick with our current chain. This is true for all Ouroboros consensus protocols, and we define it once and for all:

```
preferCandidate ::
     ConsensusProtocol p
  => proxy      p
  -> SelectView p  -- ^ Tip of our chain
  -> SelectView p  -- ^ Tip of the candidate
  -> Bool
preferCandidate _ ours cand = cand > ours
```

4.2.2 Ledger view

We mentioned in section 2.1.2 that some consensus protocols may require limited information from the ledger; for instance, the Praos consensus protocol needs access to the stake distribution for the leadership check. In the `ConsensusProtocol` abstraction, this is modelled as a *view* on the ledger state

```
type family LedgerView p :: Type
```

The ledger view will be required in only one function: when we "tick" the state of the consensus protocol. We will discuss this state management in more detail next.

4.2.3 Protocol state management

Each consensus protocol has its own type chain dependent state[7]

```
type family ChainDepState p :: Type
```

The state must be updated with each block that comes in, but just like for chain selection, we don't work with a concrete block type but instead define a *view* on blocks that is used to update the consensus state:

```
type family ValidateView p :: Type
```

We're referring to this as the `ValidateView` because updating the consensus state also serves as *validation* of (that part of) the block; consequently, validation can also *fail*, with protocol specific error messages:

[7]We are referring to this as the "chain dependent state" to emphasise that this is state that evolves with the chain, and indeed is subject to rollback when we switch to alternatives forks. This distinguishes it from chain *independent* state such as evolving private keys, which are updated independently from blocks and are not subject to rollback.

```
type family ValidationErr p :: Type
```

Updating the chain dependent state now comes as a pair of functions. As for the ledger (section 2.1.2), we first *tick* the protocol state to the appropriate slot, passing the already ticked ledger view as an argument:[8]

```
tickChainDepState ::
     ConsensusConfig p
  -> Ticked (LedgerView p)
  -> SlotNo
  -> ChainDepState p
  -> Ticked (ChainDepState p)
```

As an example, the Praos consensus protocol (section 4.6) derives its randomness from the chain itself. It does that by maintaining a set of random numbers called *nonces*, which are used as seeds to pseudo-random number generators. Every so often the current nonce is swapped out for a new one; this does not depend on the specific block, but merely on a certain slot number being reached, and hence is an example of something that the ticking function should do.

The (validation view on) a block can then be applied to the already ticked protocol state:

```
updateChainDepState ::
     ConsensusConfig       p
  -> ValidateView          p
  -> SlotNo
  -> Ticked (ChainDepState p)
  -> Except (ValidationErr p) (ChainDepState p)
```

Finally, there is a variant of this function that can we used to *reapply* a known-to-be-valid block, potentially skipping expensive cryptographic checks, merely computing what the new state is:

```
reupdateChainDepState ::
     ConsensusConfig       p
  -> ValidateView          p
  -> SlotNo
  -> Ticked (ChainDepState p)
  -> ChainDepState         p
```

Re-applying previously-validated blocks happens when we are replaying blocks from the immutable database when initialising the in-memory ledger state (section 10.2.2). It is also useful during chain selection (chapter 11): depending on the consensus protocol, we may end up switching relatively frequently between short-lived forks; when this happens, skipping expensive checks can improve the performance of the node.

How does this relate to the best case == worst case thing? Or to the asymptotic attacker/defend costs?

4.2.4 Leader selection

The final responsibility of the consensus protocol is leader selection. First, it is entirely possible for nodes to track the blockchain without ever producing any blocks themselves; indeed, this will be the case for the majority of nodes[9] In order for a node to be able to lead at all, it may need access to keys and other configuration data; the exact nature of what is required is different from protocol to protocol, and so we model this as a type family

```
type family CanBeLeader p :: Type
```

[8] Throughout the consensus layer, the result of ticking is distinguished from the unticked value at the type level. This allows to store additional (or indeed, less) information in the ticked ledger state, but also clarifies ordering. For example, it is clear in `tickChainDepState` that the ledger view we pass as an argument is already ticked, as opposed to the *old* ledger view.

[9] Most "normal" users will not produce blocks themselves, but instead delegate their stake to stakepools who produce blocks on their behalf.

A value of `CanBeLeader` merely indicates that the node has the required configuration to lead at all. It does *not* necessarily mean that the node has the right to lead in any particular slot; *this* is indicated by a value of type `IsLeader`:

```
type family IsLeader p :: Type
```

In simple cases `IsLeader` can just be a unit value ("yes, you are a leader now") but for more sophisticated consensus protocols such as Praos this will be a cryptographic proof that the node indeed has the right to lead in this slot. Checking whether a that *can* lead *should* lead in a given slot is the responsibility of the final function in this class:

```
checkIsLeader ::
     ConsensusConfig       p
  -> CanBeLeader           p
  -> SlotNo
  -> Ticked (ChainDepState p)
  -> Maybe (IsLeader       p)
```

4.3 Connecting a block to a protocol

Although a single consensus protocol might be used with many blocks, any given block is designed for a *single* consensus protocol. The following type family witnesses this relation:[10]

```
type family BlockProtocol blk :: Type
```

Of course, for the block to be usable with that consensus protocol, we need functions that construct the `SelectView` (section 4.2.1) and `ValidateView` (section 4.2.3) projections from that block:

```
class (..) => BlockSupportsProtocol blk where
  validateView ::
       BlockConfig blk
    -> Header blk -> ValidateView (BlockProtocol blk)

  selectView ::
       BlockConfig blk
    -> Header blk -> SelectView (BlockProtocol blk)
```

The `BlockConfig` is the static configuration required to work with blocks of this type; it's just another data family:

```
data family BlockConfig blk :: Type
```

4.4 Design decisions constraining the Ouroboros protocol family

TODO

TODO: Perhaps we should move this to conclusions; some of these requirements may only become clear in later chapters (like the forecasting range).

TODO

TODO: The purpose of this section should be to highlight design decisions we're already covering in this chapter that impose constraints on existing or future members of the Ouroboros protocol family.

For example, we at least have:

- max-K rollback, we insist that there be a maximum rollback length. This was true for Ouroboros Classic, but is not true for Praos/Genesis, nevertheless we insist on this for our design. We should say why this is so helpful for our design. We should also admit that this is a fundamental decision on liveness vs consistency, and that we're picking consistency over liveness. The Ouroboros family

[10] For a discussion about why we choose to make some type families top-level definitions rather than associate them with a type class, see section 25.2.

is more liberal and different members of that family can and do make different choices, so some adaptation of protocols in papers may be needed to fit this design decision. In particular this is the case for Genesis. We cannot implement Genesis as described since it is not compatible with a rollback limit.

- We insist that we can compare chains based only on their tips. For example even length is a property of the whole chain not a block, but we insist that chains include their length into the blocks in a verifiable way, which enables this tip-only checking. Future Ouroboros family members may need some adaptation to fit into this constraint. In particular the Genesis rule as described really is a whole chain thing. Some creativity is needed to fit Genesis into our framework: e.g. perhaps seeing it not as a chain selection rule at all but as a different (coordinated) mode for following headers.
- We insist that a strict extension of a chain is always preferred over that chain.
- We insist that we never roll back to a strictly shorter chain.
- The minimum cyclic data dependency time: the minimum time we permit between some data going onto the chain and it affecting the validity of blocks or the choices made by chain selection. This one is a constraint on both the consensus algorithm and the ledger rules. For example this constrains the Praos epoch structure, but also ledger rules like the Shelley rule on when genesis key delegations or VRF key updates take effect. We should cover why we have this constraint: arising from wanting to do header validation sufficiently in advance of block download and validation that we can see that there's a potential longer valid chain.
- The ledger must be able to look ahead sufficiently to validate $k + 1$ headers (to guarantee a roll back of k). TODO: We should discuss this in more detail.

TODO

4.5 Permissive BFT

Defined in [7] Not to be confused with "Practical BFT" [3]

4.5.1 Background

Discuss *why* we started with Permissive BFT (backwards compatible with Ouroboros Classic).

Duncan suitable section.

4.5.2 Implementation

4.5.3 Relation to the paper

Permissive BFT is a variation on Ouroboros BFT, defined in [9]. We have included the main protocol description from that paper as fig. 4.1 in this document; the only difference is that we've added a few additional labels so we can refer to specific parts of the protocol description below.

It will be immediately obvious from fig. 4.1 that this description covers significantly more than what we consider to be part of the consensus protocol proper here. We will discuss the various parts of the BFT protocol description below.

Clock update and network delivery The BFT specification requires that "with each advance of the clock (..) a collection of transactions and blockchains are pushed to the server". We consider neither block submission nor transaction submission to be within the scope of the consensus algorithm; see sections 3.1.2 and 15.4 and sections 3.1.2 and 15.3 instead, respectively.

Mempool update (item 1). The design of the mempool is the subject of chapter 13. Here we only briefly comment on how it relates to what the BFT specification assumes:

- *Consistency* (item 1a). Our mempool does indeed ensure consistency. In fact, we require something strictly stronger; see section 13.1 for details.
- *Time-to-live (TTL)* (item 1b). The BFT specification requires that transactions stay in the mempool for a maximum of u rounds, for some configurable u. Our current mempool does not have explicit support for a TTL parameter. The Shelley ledger will have support for TTL starting with the "Allegra" era, so that transactions are only valid within a certain slot window; this is part of the normal ledger rules however and requires no explicit support from the consensus layer. That's not to say that explicit support would not be useful; see section 24.3 in the chapter on future work.
- *Receipts* (item 1c). We do not offer any kind of receipts for inclusion in the mempool. Clients such as wallets must monitor the chain instead (see also [5]). The BFT specification marks this as optional so this is not a deviation.

Blockchain update (item 2). The BFT specification requires that the node prefers any valid chain over its own, as long as its strictly longer. *We do not satisfy this requirement.* The chain selection rule for Permissive BFT is indeed the longest chain rule, *but* consensus imposes a global maximum rollback (the security parameter k; section 4.1.2). In other words, nodes *will* prefer longer chains over its own, *provided* that the intersection between that chain and the nodes own chain is no more than k blocks away from the node's tip.

Justify this maximum rollback?

Moreover, our definition of validity is also different. We do require that hashes line up (item 2b), although we do not consider this part of the responsibility of the consensus protocol, but instead require this independent of the choice of consensus protocol when updating the header state (section 7.2.2). We do of course also require that the transactions in the block are valid (item 2c), but this is the responsibility of the ledger layer instead (chapter 5); the consensus protocol should be independent from what's stored in the block body.

Permissive BFT is however different from BFT *by design* in the signatures we require.[11] BFT requires that each block is signed strictly according to the round robin schedule (item 2a); the whole point of *permissive* BFT is that we relax this requirement and merely require that blocks are signed by *any* of the known core nodes.

Permissive BFT is however not *strictly* more permissive than BFT: although blocks do not need to be signed according to the round robin schedule, there is a limit on the number of signatures by any given node in a given window of blocks. When a node exceeds that threshold, its block is rejected as invalid. Currently that threshold is set to 0.22 [7, Appendix A, Calculating the t parameter], which was considered to be the smallest value that would be sufficiently unlikely to consider a chain generated by Ouroboros Classic as invalid (section 4.5.1) and yet give as little leeway to a malicious node as possible. This has an unfortunate side effect, however. BFT can always recover from network partitions [9, Section 1, Introduction], but this is not true for PBFT: in a setting with 7 core nodes (the same setting as considered in the PBFT specification), a 4:3 network partition would quickly lead to *both* partitions being unable to produce more blocks; after all, the nodes in the partition of 4 nodes would each sign 1/4th of the blocks, and the nodes in the partition of 3 nodes would each sign 1/3rd. Both partitions would therefore quickly stop producing blocks. Picking 0.25 for the threshold instead of 0.22 would alleviate this problem, and would still be conform the PBFT specification, which says that the value must be in the closed interval $[\frac{1}{5}, \frac{1}{4}]$. Since PBFT is however no longer required (the Byron era is past and fresh deployments would not need Permissive BFT but could use regular BFT), it's probably not worth reconsidering this, although it *is* relevant for the consensus tests (section 19.1).

Blockchain extension (item 3). The leadership check implemented as part of PBFT is conform specification (eq. (4.1)). The rest of this section matches the implementation, modulo some details some

[11] There is another minor deviation from the specification: we don't require an explicit signature on the block body. Instead, we have a single signature over the header, and the header includes a *hash* of the body.

of which we already alluded to above:

- The block format is slightly different; for instance, we only have a single signature (footnote 11).
- Blocks in Byron have a maximum size, so we cannot necessarily take *all* valid transactions from the mempool.
- Block diffusion is not limited to the suffix of the chain: clients can request *any* block that's on the chain. This is of course critical to allow nodes to join the network later, something which the BFT paper does not consider.

It should also be pointed out that we consider neither block production nor block diffusion to be part of the consensus protocol at all; only the leadership check itself is.

Ledger reporting . Although we do offer a way to query the state of the ledger (section 5.3), we do not offer a query to distinguish between finalised/pending blocks. TODO: It's also not clear to me why the BFT specification would consider a block to be finalised as soon as it's $3t+1$ blocks deep (where t is the maximum number of core nodes). The paper claims that BFT can always recover from a network partition, and the chain selection rule in the paper requires supporting infinite rollback. TODO

4.6 Praos

TODO: Discuss Δ: When relating the papers to the implementation, we loosely think of Δ as roughly having value 5, i.e., there is a maximum message delay of 5 slots. However, this link to the paper is tenuous at best: the messages the paper expects the system to send, and the messages that the system *actually* sends, are not at all the same. Defining how these relate more precisely would be critical for a more formal statement of equivalence between the paper and the implementation, but such a study is well outside the scope of this report.

4.6.1 Active slot coefficient

4.6.2 Implementation

4.6.3 Relation to the paper

[1]

4.7 Combinator: Override the leader schedule

Parameters:

n	total number of core nodes
t	maximum number of core nodes
	(we do not make this distinction between n and t in the consensus layer, effectively setting $n = t$)
u	time to live (TTL) of a transaction

Protocol:

The i-th server locally maintains a blockchain $B_0 B_1 \dots B_l$, an ordered sequence of transactions called a mempool, and carries out the following protocol:

Clock update and network delivery With each advance of the clock to a slot sl_j, a collection of transactions and blockchains are pushed to the server by the network layer. Following this, the server proceeds as follows:

1. **Mempool update**.
   (a) Whenever a transaction tx is received, it is added to the mempool as long as it is consistent with
      i. the existing transactions in the mempool and
      ii. the contents of the local blockchain.
   (b) The transaction is maintained in the mempool for u rounds, where u is a parameter.
   (c) Optionally, when the transaction enters the mempool the server can return a signed receipt back to the client that is identified as the sender.
2. **Blockchain update**. Whenever the server becomes aware of an alternative blockchain $B_0 B'_1 \dots B'_s$ with $s > l$, it replaces its local chain with this new chain provided it is valid, i.e. each one of its blocks $(h, d, sl_j, \sigma_{sl}, \sigma_{\text{block}})$
   (a) contains proper signatures
      i. one for time slot sl_j and
      ii. one for the entire block

      by server i such that $i - 1 = (j - 1) \bmod n$
   (b) h is the hash of the previous block, and
   (c) d is a valid sequence of transactions w.r.t. the ledger defined by the transactions found in the previous blocks
3. **Blockchain extension**. Finally, the server checks if it is responsible to issue the next block by testing if

$$i - 1 = (j - 1) \bmod n \tag{4.1}$$

   In such case, this i-th server is the slot leader. It

   - collects the set d of all valid transactions from its mempool and
   - appends the block $B_{l+1} = (h, d, sl_j, \sigma_{sl}, \sigma_{\text{block}})$ to its blockchain, where

$$\begin{aligned} \sigma_{sl} &= \mathsf{Sign}_{\mathsf{sk}_i}(sl_j) \\ \sigma_{\text{block}} &= \mathsf{Sign}_{\mathsf{sk}_i}(h, d, sl_j, \sigma_{sl}) \\ h &= H(B_l) \end{aligned}$$

     It then diffuses B_{l+1} as well as any requested blocks from the suffix of its blockchain that covers the most recent $2t + 1$ slots.

Ledger Reporting Whenever queried, the server reports as "finalised" the ledger of transactions contained in the blocks $B_0 \dots B_m, m \leq l$, where B_m has a slot time stamp more than $3t + 1$ slots in the past. Blocks $B_{m+1} \dots B_l$ are reported as "pending".

Figure 4.1: Ouroboros-BFT [9, Figure 1]

4.8 Separation of responsibility between consensus and ledger

4.8.1 Vision

In the vision that underlies the abstract design of the consensus layer, the separation of responsibility between the consensus layer and the ledger layer happens along three axes.

Block *selection* versus block *contents*

The primary objective of the consensus layer is to ensure that *consensus* is reached: that is, everyone agrees on (a sufficiently long prefix of) the chain. From a sufficiently high vantage point, the consensus layer could reasonably be described as an implementation of the various Ouroboros papers (Praos, Genesis, etc.). A critical component of this is *chain selection*, choosing between competing chains. The consensus layer does not need to be aware of what is inside the blocks that it is choosing between.

By contrast, the ledger layer is not aware of multiple chains at all, and will never need to execute chain selection: it exclusively deals with linear histories. *Its* primary objective is to define the contents of blocks, along with rules that interpret those contents, computing the *ledger state*.

Construction versus *verification*

The ledger layer only ever deals with fully formed blocks. Its responsibility is to *verify* those blocks and describe how they transform the ledger state. But those blocks need to come from somewhere in the first place; block *construction* is the responsibility of the consensus layer. This dichotomy manifests itself in two ways:

- When the ledger layer verifies a block, it must verify whether or not the node that produced the block had the right to do so, that is, whether or not it was a slot leader in the block's slot (though it may be argued that this should be a consensus concern instead, see below). Typically, it will need only access to the node's *public* key to do so. Note that multiple nodes may have the right to produce a block in any given slot; the ledger layer is not checking for *the* slot leader, but rather for *a* slot leader.

  By contrast, the consensus layer is not checking if *some* node is a leader for slot, but rather whether *it* is a leader for the current slot, and if so, produce a block for that slot (along with evidence that it had the right to do so). Typically, it will need access to the node's *private* key in order to execute that check.
- Blocks are only valid with respect to a particular ledger state. Since blocks specify their predecessor (the predecessor hash), they also implicitly specify which ledger state they should be evaluated against: the ledger state that was the result of applying that predecessor block (or the genesis ledger state for the very first block).

  By contrast, when the consensus layer produces a block, it must construct a block that is valid with respect to the node's *current* ledger state, and *choose* the predecessor of that new block to be the tip of the node's current chain.

Stateful versus *stateless*

The ledger layer is entirely stateless: it is a pure function that accepts a ledger state and a block as input and produces the new ledger state (or an error if the block was invalid). State management is the responsibility of the consensus layer:

- The consensus layer must maintain the current ledger state, and pass that to the ledger layer when validating blocks that fit neatly onto the node's current tip. In addition, the consensus must provide efficient access to *historical* ledger states, so that it can validate (and possibly adopt) alternative forks of the chain.

- Although the consensus layer does not need to be aware of the exact nature of the block contents, it *does* have to collect these "transactions" and consider them when producing a block (the mempool, chapter 13). If it chooses to do eager transaction validation (that is, before it actually produces a block), it will need support from the ledger layer to do; when producing a block, it will need assistance from the ledger layer to produce the block body.

  In both cases (transaction validation and block body construction), the consensus layer is responsible for passing an appropriate ledger state to the ledger layer. In the case of block production, the choice of ledger state is clear: the current ledger state. For the mempool, it is slightly less clear-cut; the mempool is effectively constructing a "virtual" block with a predecessor chosen from the node's current chain, near its tip.[12]

Ideally, the implementation of a particular consensus protocol (say, Praos) should be usable with any choice of ledger (cryptocurrency or otherwise), and conversely, a particular ledger (say, Shelley) should be usable with any choice of consensus protocol (Praos, Genesis, or indeed a different consensus protocol entirely). The consensus protocol *does* need some limited information from the ledger, but we can provide this separately (section 5.1.4), and abstract over what a particular consensus algorithm needs from the ledger layer it is used with (specifically, the `SelectView` and the `LedgerView`, discussed in sections 4.2.1 and 4.2.2).

4.8.2 Practice

In practice, the separation is not quite so clean. Partly this is for historical reasons. When the Cardano blockchain was re-implemented, the new consensus layer and the new ledger layer were developed in tandem, and it was not always practical to have one wait for design decisions by the other. For example, most of Praos is currently implemented in the ledger layer, despite the ledger layer never having to do chain selection, ever. These are issues that we can resolve with some relatively minor refactoring.

More problematic is that the current ledgers are not designed to be parametric in a choice of consensus algorithm. Specifically, the Shelley ledger hardcodes Praos. At some level, that statement makes no sense: the ledger layer never needs to execute chain selection nor decide if it's a leader for a given slot. However, the *verification* of a block by the ledger layer currently includes verification of the cryptographic proof produced by the consensus layer when constructing a block. This is specific to Praos; other consensus algorithms may require entirely different fields in the block (header). So while the ledger layer is morally independent from the choice of consensus algorithm, in practice it includes just enough information that running it with a different consensus algorithm is difficult to do.

In an ideal world, the Shelley ledger would not be aware of the consensus algorithm at all. Since the implementation of the consensus protocol, and details of the fields required in blocks to support that protocol, are the responsibility of the consensus layer, it would make sense to move the leader verification check from the ledger layer into the consensus header check instead. Block assembly now becomes more of a joint effort between the consensus layer and the ledger layer: the ledger layer produces the block body and some fields in the block header[13]), whereas the consensus layer produces the fields in the block header that are required by the consensus protocol.

Unfortunately, disentangling the two isn't *quite* that easy. In particular, the Shelley ledger supports key delegation, which is affecting the leadership check. Disentangling this would be non-trivial; it's not clear what consensus-protocol independent delegation would even *mean* and what kind of data it should carry. Solving this will probably require some parameterization in the *other* direction, with the ledger rules for delegation allowing for some protocol specific data to be included.

[12] We could update this "virtual" block every time that the node's current chain changes, so that the virtual block's predecessor is always the current chain's tip. This however couples two concurrent processes more tightly than required, and is moreover costly: re-evaluating the mempool can be expensive.

[13] In an perfect world this header/body boundary would align neatly with the consensus/ledger boundary; I think this ought to be possible in principle, but in the current design this is non-trivial to achieve, since the ledger layer is interpreting some fields in the header; for example, it is executing some rules in response to epoch transitions, which it detects based on fields in the header.

Chapter 5

Interface to the ledger

5.1 Abstract interface

In section 2.1.2 we identified three responsibilities for the ledger layer:

- "Ticking" the ledger state, applying any time related changes (section 5.1.1). This is independent from blocks, both at the value level (we don't need a block in order to tick) and at the type level.
- Applying and verifying blocks (section 5.1.2). This obviously connects a ledger and a block type, but we try to avoid to talk about *the* ledger corresponding to a block, in order to improve compositionality; we will see examples of where this comes in useful in the definition of the extended ledger state (section 7.2.2) and the ledger database (chapter 10).
- Projecting out the ledger view (section 5.1.4), connecting a ledger to a consensus protocol.

We will discuss these responsibilities one by one.

5.1.1 Independent definitions

We will start with ledger API that can be defined independent of a choice of block or a choice of consensus protocol.

Configuration

Like the other abstractions in the consensus layer, the ledger defines its own type of required static configuration

```
type family LedgerCfg l :: Type
```

Tip

We require that any ledger can report its tip as a `Point`. A `Point` is either genesis (no blocks have been applied yet) or a pair of a hash and slot number; it is parametric over *l* in order to allow different ledgers to use different hash types.

```
class GetTip l where
  getTip :: l -> Point l
```

Ticking

We can now define the `IsLedger` class as

```
class (GetTip l, GetTip (Ticked l), ..) => IsLedger l where
  type family LedgerErr l :: Type
  applyChainTick :: LedgerCfg l -> SlotNo -> l -> Ticked l
```

The type of `applyChainTick` is similar to the type of `tickChainDepState` we saw in section 4.2.3. Examples of the time-based changes in the ledger state include activating delegation certificates in the Byron ledger, or paying out staking rewards in the Shelley ledger.

Ticking is not allowed to fail (it cannot return an error). Consider what it would mean if it *could* fail: it would mean that a previous block was accepted as valid, but set up the ledger state so that no matter what would happen next, as soon as a particular moment in time is reached, the ledger would fail to advance any further. Obviously, such a situation cannot be permitted to arise (the block should have been rejected as invalid).

Note that ticking does not change the tip of the ledger: no blocks have been applied (yet). This means that we should have

$$\texttt{getTip}\ l = \texttt{getTip}\ (\texttt{applyChainTick}_{cfg}\ s\ l) \tag{5.1}$$

Ledger errors

The inclusion of `LedgerErr` in `IsLedger` is perhaps somewhat surprising. `LedgerErr` is the type of errors that can arise when applying blocks to the ledger, but block application is not yet defined here. Nonetheless, a ledger can only be used with a *single* type of block[14], and consequently can only have a *single* type of error; the only reason block application is defined separately is that a single type of *block* can be used with multiple ledgers (in other words, this is a 1-to-many relationship).[15]

5.1.2 Applying blocks

If `applyChainTick` was analogous to `tickChainDepState`, then `applyLedgerBlock` and `reapplyLedgerBlock` are analogous to `updateChainDepState` and `reupdateChainDepState`, respectively (section 4.2.3): apply a block to an already ticked ledger state:

```
class (IsLedger l, ..) => ApplyBlock l blk where
  applyLedgerBlock ::
    LedgerCfg l -> blk -> Ticked l -> Except (LedgerErr l) l
  reapplyLedgerBlock ::
    LedgerCfg l -> blk -> Ticked l -> l
```

The discussion of the difference between, and motivation for, the distinction between application and reapplication in section 4.2.3 about the consensus protocol state applies here equally.

5.1.3 Linking a block to its ledger

We mentioned at the start of section 5.1 that a single block can be used with multiple ledgers. Nonetheless, there is one "canonical" ledger for each block; for example, the Shelley block is associated with the Shelley ledger, even if it can also be applied to the extended ledger state or the ledger DB. We express this through a data family linking a block to its "canonical ledger state":

[14]While it is true that the Cardano ledger can be used with Byron blocks, Shelley blocks, Goguen blocks, etc., this distinction between the different blocks is invisible to most of the consensus layer. The whole raison d'être of the hard fork combinator (chapter 16) is to present a composite ledger (say, the one consisting of the Byron, Shelley and Goguen eras) as a single type of ledger with a single type of block. The rest of the consensus layer is not aware that this composition has happened; from its point perspective it's just another ledger with an associated block type.

[15]Defining `LedgerErr` in `ApplyBlock` (section 5.1.2) would result in ambiguous types, since it would not refer to the `blk` type variable of that class.

```
data family LedgerState blk :: Type
```

and then require that it must be possible to apply a block to its associated ledger state

```
class ApplyBlock (LedgerState blk) blk => UpdateLedger blk
```

(this is an otherwise empty class). For convenience, we then also introduce some shorthand:

```
type LedgerConfig      blk = LedgerCfg (LedgerState blk)
type LedgerError       blk = LedgerErr (LedgerState blk)
type TickedLedgerState blk = Ticked    (LedgerState blk)
```

5.1.4 Projecting out the ledger view

In section 2.1.2 we mentioned that a consensus protocol may require some information from the ledger, and in section 4.2.2 we saw that this is modelled as the `LedgerView` type family in the `ConsensusProtocol` class. A ledger and a consensus protocol are linked through the block type (indeed, apart from the fundamental concepts we have discussed so far, most of consensus is parameterised over blocks, not ledgers or consensus protocols). Recall from section 4.3 that the `BlockProtocol` type family defines for each block what the corresponding consensus protocol is; we can use this to define the projection of the ledger view (defined by the consensus protocol) from the ledger state as follows:

```
class (..) => LedgerSupportsProtocol blk where
  protocolLedgerView ::
       LedgerConfig blk
    -> Ticked (LedgerState blk)
    -> Ticked (LedgerView (BlockProtocol blk))

  ledgerViewForecastAt ::
       LedgerConfig blk
    -> LedgerState blk
    -> Forecast (LedgerView (BlockProtocol blk))
```

The first method extracts the ledger view out of an already ticked ledger state; think of it as the "current" ledger view. Forecasting deserves a more detailed discussion and will be the topic of the next section.

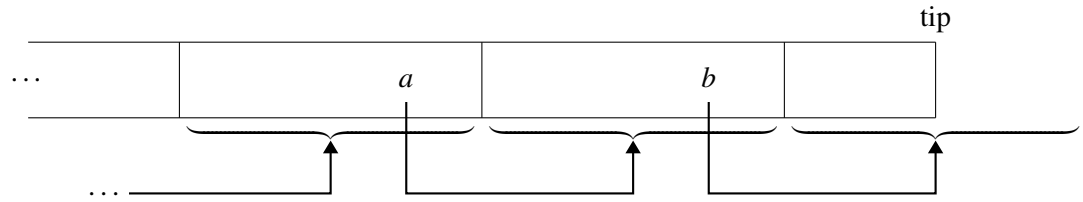
5.2 Forecasting

5.2.1 Introduction

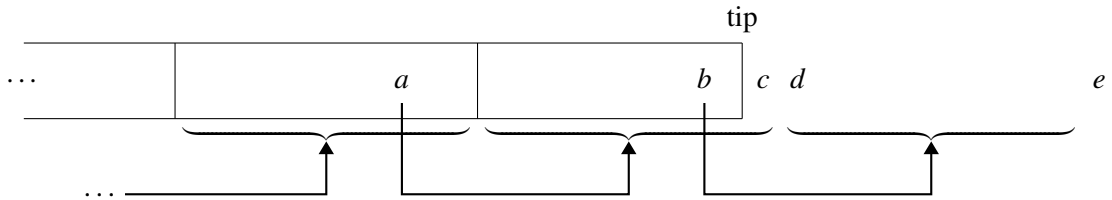
In section 3.1.1 we discussed the need to validate headers from upstream peers. In general, header validation requires information from the ledger state. For example, in order to verify whether a Shelley header was produced by the right node, we need to know the stake distribution (recall that in Shelley the probability of being elected a leader is proportional to the stake); this information is precisely what is captured by the `LedgerView` (section 4.2.2). However, we cannot update the ledger state with block headers only, we need the block bodies: after all, to stay with the Shelley example, the stake evolves based on the transactions that are made, which appear only in the block bodies.

Not all is lost, however. The stake distribution used by the Shelley ledger for the sake of the leadership check *is not the* current *stake distribution*, but the stake distribution as it was at a specific point in the past. Moreover, that same stake distribution is then used for all leadership checks in a given period of time.[16] In the depiction below, the stake distribution as it was at point b is used for the leadership checks near the current tip, the stake distribution at point a was used before that, and so forth:

[16]The exact details of precisely *how* the chain is split is not relevant to the consensus layer, and is determined by the ledger layer.



This makes it possible to *forecast* what the stake distribution (i.e., the ledger view) will be at various points. For example, if the chain looks like



then we can "forecast" that the stake distribution at point c will be the one established at point a, whereas the stake distribution at point d will be the one established at point b. The stake distribution at point e is however not yet known; we say that e is "out of the forecast range".

5.2.2 Code

Since we're always forecasting what the ledger would look like *if it would be advanced to a particular slot*, the result of forecasting is always something ticked:[17]

```
data Forecast a = Forecast {
      forecastAt  :: WithOrigin SlotNo
    , forecastFor :: SlotNo -> Except OutsideForecastRange (Ticked a)
    }
```

Here `forecastAt` is the tip of the ledger in which the forecast was constructed and `forecastFor` is constructing the forecast for a particular slot, possibly returning an error message of that slot is out of range. This terminology—a forecast constructed *at* a slot and computed *for* a slot—is used throughout both this technical report as well as the consensus layer code base.

5.2.3 Ledger view

For the ledger view specifically, the `LedgerSupportsProtocol` class (section 5.1.4) requires a function

```
ledgerViewForecastAt ::
     LedgerConfig blk
  -> LedgerState blk
  -> Forecast (LedgerView (BlockProtocol blk))
```

This function must satisfy two important properties:

Sufficient range When we validate headers from an upstream node, the most recent usable ledger state we have is the ledger state at the intersection of our chain and the chain of the upstream node. That intersection will be at most k blocks back, because that is our maximum rollback and we disconnect from nodes that fork from our chain more than k blocks ago (section 4.1.2). Furthermore, it is only useful to track an upstream peer if we might want to adopt their blocks, and we only switch to their chain if it is longer than ours (section 4.1.1). This means that in the worst case scenario, with

[17] An *unticked* ledger view would arise from deriving the ledger view from the *current* ledger state, not taking (nor needing to take) into account any changes that have been scheduled for later slots. The unticked ledger view is however rarely useful; when we validate a header, any changes that have been scheduled in the most recent ledger state for slots before or at the slot number of the header must be applied before we validate the header; we therefore almost exclusively work with ticked ledger views.

the intersection k blocks back, we need to be able to evaluate $k + 1$ headers in order to adopt the alternative chain. However, the range of a forecast is based on *slots*, not blocks; since not every slot may contain a block (section 17.2), the range needs to be sufficient to *guarantee* to contain at least $k + 1$ blocks[18]; we will come back to this in section 24.4.

The network layer may have additional reasons for wanting a long forecast range; see section 3.1.1.

Relation to ticking Forecasting is not the only way that we can get a ledger view for a particular slot; alternatively, we can also *tick* the ledger state, and then ask for the ledger view at that ticked ledger state. These two ways should give us the same answer:

$$\begin{array}{lllll} \text{whenever} & \texttt{forecastFor}\ (\texttt{ledgerViewForecastAt}_{cfg}\ l)\ s & = & \texttt{Right} & l' \\ \text{then} & \texttt{protocolLedgerView}_{cfg}\ (\texttt{applyChainTick}_{cfg}\ s\ l) & = & & l' \end{array} \tag{5.2}$$

In other words, whenever the ledger view for a particular slot is within the forecast range, then ticking the ledger state to that slot and asking for the ledger view at the tip should give the same answer. Unlike forecasting, however, ticking has no maximum range. The reason is the following fundamental difference between these two concepts:

> **(Forecast vs. ticking)** When we *forecast* a ledger view, we are predicting what that ledger view will be, *no matter which blocks will be applied to the chain* between the current tip and the slot of the forecast. By contrast, when we *tick* a ledger, we are applying any time-related changes to the ledger state in order to apply the *next* block; in other words, when we tick to a particular slot, *there* are *no blocks in between the current tip and the slot we're ticking to*. Since there are no intervening blocks, there is no uncertainty, and hence no limited range.

5.3 Queries

5.4 Abandoned approach: historical states

[18] Due to a misalignment between the consensus requirements and the Shelley specification, this is not the case for Shelley, where the effective maximum rollback is in fact $k - 1$; see section B.2).

Chapter 6

Serialisation abstractions

Some of the various pieces of data that are handled by consensus also need to be serialised to a binary format so that they can be:

1. written/read to/from *storage* (see chapter 7) or;
2. sent/received across the *network* (e.g., headers via the chain sync protocol chapter 14).

The two serialisation purposes above have different requirements and are independent of each other. For example, when establishing a network connection, a version number is negotiated. We can vary the network serialisation format depending on the version number, allowing for instance to include some more information in the payload. A concrete example of this is that starting from a certain version, we include the block size in the payload when sending a Byron header across the network as the header itself does not contain it. This kind of versioning only concerns the network and is independent of the storage layer. Hence we define separate abstractions for them, decoupling them from each other.

Can I talk about Byron here?

For both abstractions, we use the CBOR (Concise Binary Object Representation) format, because it has the following benefits, paraphrasing the `cborg` library:

command for acronyms?

link?

- fast serialisation and deserialisation
- compact binary format
- stable format across platforms (32/64bit, big/little endian)
- potential to read the serialised format from other languages
- incremental or streaming (de)serialisation
- suitable to use with untrusted input (resistance to asymmetric resource consumption attacks)
- ...

Moreover, CBOR was chosen for the initial implementation of the Cardano blockchain, with which we must maintain binary compatibility. While it was possible to switch to another format for the block types developed after the initial implementation, we saw no reason to switch.

correct?

We will now discuss both serialisation abstractions in more detail.

6.1 Serialising for storage

The following data is stored on disk (see chapter 7):

- Blocks

- The extended ledger state (see section 7.2.2 and section 10.2) which is the combination of:
  - The header state (section 7.2.2)
  - The ledger state — link?

We use the following abstraction for serialising data to and from disk:

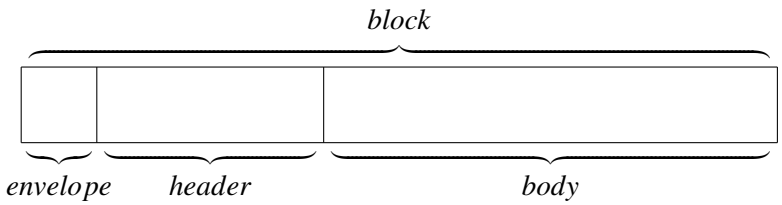
```
class EncodeDisk blk a where
  encodeDisk :: CodecConfig blk -> a -> Encoding

class DecodeDisk blk a where
  decodeDisk :: CodecConfig blk -> forall s. Decoder s a
```

- These type classes have two type parameters: the block `blk`, over which most things are parameterised, and `a`, the type to (de)serialise. For example, `a` can be the block type itself or the type corresponding to the ledger state.
- `CodecConfig blk` is a data family that defines the extra configuration needed for (de)serialisation. For example, to deserialise an EBB (chapter 23), the number of slots per epoch needs to be known statically to compute the slot of the block based on the epoch number, as the serialisation of an EBB does not contain its slot number, but the in-memory representation does. This configuration is kept as small as possible and is ideally empty.
- The `a -> Encoding` and `forall s. Decoder s a` are the types for respectively encoders and decoders of the `cborg` library. — link?
- The encoder and decoder are split in two classes because they are not always *symmetric*: the instantiation of `a` in the encoder is not always the same as in the corresponding decoder. This is because blocks are *annotated* with their serialisation. We discuss this in more detail in section 6.3.

6.1.1 Nested contents

By writing a block to disk we automatically have written the block's header to disk, as the header is a part of the block. While we never write just a header, we do support *reading* just the header. This is more efficient than reading the entire block and then extracting the header, as fewer bytes have to be read from disk and deserialised.



Extracting the header from a block on disk can be very simple, like in the figure above. The block starts with an envelope, which is followed by the block header and the block body. In this case, we read the bytes starting from the start of the header until the end of the header, which we then decode. We use the following abstraction to represent this information:

```
data BinaryBlockInfo = BinaryBlockInfo {
      headerOffset :: !Word16
    , headerSize   :: !Word16
    }

class HasBinaryBlockInfo blk where
  getBinaryBlockInfo :: blk -> BinaryBlockInfo
```

As the size of a header can vary on a per-block basis, we maintain this information *per block* in the storage layer. We trade four extra bytes of storage and memory space for faster reading of headers. link?

However, it is not for every type of block the case that the encoding of a header can literally be sliced out of the encoding of the corresponding block. The serialisation of a header when embedded in a block might be different from the serialisation of a header on its own. For example, the standalone header might require an additional envelope or a different one than the block's envelope.

A concrete example of this are the Byron blocks and headers. A Byron block is either a regular block or an epoch boundary block (EBB) (discussed in chapter 23). A regular block has a different header than an EBB, consequently, their encoding differs. The envelope of the encoding of a Byron block includes a tag indicating whether the block is a regular block or an EBB, so that the decoder knows what kind of header and body to expect. For the same reason, the envelope of the encoding of a standalone Byron header includes the same tag. However, when we slice out the header from the Byron block and feed that to the decoder for Byron headers, the envelope containing the tag will be *missing*.

The same problem presents itself for the hard fork combinator (chapter 16): when using the hard fork combinator to combine two block types, A and B, into one, the block's envelope will (typically) indicate whether it is a block of type A or B. The header corresponding to such a block will have a similar envelope. When we slice the header out of such a block, the required envelope will be missing. The right envelope has to be prepended so that the header decoder knows whether it should expect A or B.

The header is *nested* inside the block and to be able to decode it, we need some more *context*, i.e., the envelope of the header. In the storage layer (chapter 7), we store the context of each block in an index (in-memory or on-disk, depending on the database) so that after reading both the context and the sliced header, we can decode the header without having to read and decode the entire block. We capture this idea in the following abstractions.

```
data family NestedCtxt_ blk :: (Type -> Type) -> (Type -> Type)
```

As usual, we parameterise over the block type. We also parameterise over another functor, e.g., `f`, which in practice will be instantiated to `Header`, but in the future, there might be more types of nested contents, other than headers, e.g., block bodies. The constructors of this data family will represent the different types of context available, e.g., for Byron a context for regular blocks and a context for EBBs.

`NestedCtxt` is indexed by `blk`: it is the block that determines this type. However, we often want to partially apply the second argument (the functor), leaving the block type not yet defined, hence we define:

```
newtype NestedCtxt f blk a = NestedCtxt {
      flipNestedCtxt :: NestedCtxt_ blk f a
    }
```

The `a` type index will correspond to the raw, sliced header that requires the additional context. It can vary with the context, e.g., the context for a Byron EBB will fix `a` to a raw EBB header (without the necessary envelope).

Now that we have defined `NestedCtxt`, we can define the class that allows us to separate the nested type (the header) into the context and the raw, sliced type (the raw header, `a`), as well as the inverse:

```
class (..) => HasNestedContent f blk where
  unnest :: f blk -> DepPair (NestedCtxt f blk)
  nest   :: DepPair (NestedCtxt f blk) -> f blk
```

`DepPair` is a dependent pair that allows us to hide the type parameter `a`. When writing a block, `unnest` is used to extract the context so that it can be stored in the appropriate index. When reading a header, `nest` is used to combine the context, read from the appropriate index, with the raw header into the header.

In certain scenarios, we do not have access to the separately stored context of the block, but we do have access to the encoded block, in which case we should be able to able to extract the context directly from the encoded block, without having to decode it entirely. We use the `ReconstructNestedCtxt` class for this:

```
class HasNestedContent f blk => ReconstructNestedCtxt f blk where
  reconstructPrefixLen  :: proxy (f blk) -> PrefixLen
```

```
  reconstructNestedCtxt ::
       proxy (f blk)
    -> ShortByteString
    -> ..
    -> SomeSecond (NestedCtxt f) blk
```

The `PrefixLen` is the number of bytes extracted from the beginning of the encoded block required to reconstruct the context. The `ShortByteString` corresponds to these bytes. The `reconstructNestedCtxt` method will parse this bytestring and return the corresponding context. The `SomeSecond` type is used to hide the type parameter `a`.

As these contexts and context-dependent types do not fit the mould of the `EncodeDisk` and `DecodeDisk` classes described in section 6.1, we define variants of these classes:

```
class EncodeDiskDepIx f blk where
  encodeDiskDepIx :: CodecConfig blk
                  -> SomeSecond f blk -> Encoding

class DecodeDiskDepIx f blk where
  decodeDiskDepIx :: CodecConfig blk
                  -> Decoder s (SomeSecond f blk)

class EncodeDiskDep f blk where
  encodeDiskDep :: CodecConfig blk -> f blk a
                -> a -> Encoding

class DecodeDiskDep f blk where
  decodeDiskDep :: CodecConfig blk -> f blk a
                -> forall s. Decoder s (ByteString -> a)
```

explain?

6.2 Serialising for network transmission

The following data is sent across the network:

- Header hashes
- Blocks
- Headers
- Transactions
- Transaction IDs
- Transaction validation errors
- Ledger queries
- Ledger query results

less whitespace

We use the following abstraction for serialising data to and from the network:

```
class SerialiseNodeToNode blk a where
  encodeNodeToNode :: CodecConfig blk
                   -> BlockNodeToNodeVersion blk
                   -> a -> Encoding
  decodeNodeToNode :: CodecConfig blk
                   -> BlockNodeToNodeVersion blk
                   -> forall s. Decoder s a
```

```
class SerialiseNodeToClient blk a where
  encodeNodeToClient :: CodecConfig blk
                     -> BlockNodeToClientVersion blk
                     -> a -> Encoding
  decodeNodeToClient :: CodecConfig blk
                     -> BlockNodeToClientVersion blk
                     -> forall s. Decoder s a
```

These classes are similar to the ones used for storage (section 6.1), but there are some important differences:

- The encoders and decoders are always symmetric, which means we do not have to separate encoders from decoders and can merge them in a single class. Nevertheless, some of the types sent across the network still have to deal with annotations (section 6.3), we discuss how we solve this in section 6.2.2.
- We have separate classes for *node-to-node* and *node-to-client* serialisation. By separating them, we are more explicit about which data is serialised for which type of connection. Node-to-node protocols and node-to-client protocols have different properties and requirements. This also gives us the ability to, for example, use a different encoding for blocks for node-to-node protocols than for node-to-client protocols. link?
- The methods in these classes all take a *version* as argument. We will discuss versioning in section 6.2.1.

6.2.1 Versioning

As requirements evolve, features are added, data types change, constructors are added and removed. For example, adding the block size to the Byron headers, adding new ledger query constructors, etc. This affects the data we send across the network. In a distributed network of nodes, it is a given that not all nodes will simultaneously upgrade to the latest released version and that nodes running different versions of the software, i.e., different versions of the consensus layer, will try to communicate with each other. They should of course be able to communicate with each other, otherwise the different versions would cause partitions in the network.

This means we should be careful to maintain binary compatibility between versions. The network layer is faced with the same issue: as requirements evolve, network protocols (block fetch, chain sync) are modified change (adding messages, removing messages, etc.), network protocols are added or retired, etc. While the network layer is responsible for the network protocols and the encoding of their messages, the consensus layer is responsible for the encoding of the data embedded in these messages. Changes to either should be possible without losing compatibility: a node should be able to communicate successfully with other nodes that run a newer or older version of the software, up to certain limits (old versions can be retired eventually). link?

To accomplish this, the network layer uses *versions*, one for each bundle of protocols:

```
data NodeToNodeVersion
    = NodeToNodeV_1
    | NodeToNodeV_2
    | ..

data NodeToClientVersion
    = NodeToClientV_1
    | NodeToClientV_2
    | ..
```

For each backwards-incompatible change, either a change in the network protocols or in the encoding of the consensus data types, a new version number is introduced in the corresponding version data type.

When the network layer establishes a connection with another node or client, it will negotiate a version number during the handshake: the highest version that both parties can agree on. This version number is then passed to any client and server handlers, which decide based on the version number which protocols to start and which protocol messages (not) to send. A new protocol message would only be sent when the version number is greater or equal than the one with which it was introduced.

This same network version is passed to the consensus layer, so we can follow the same approach. However, we decouple the network version numbers from the consensus version numbers for the following reason. A new network version number is needed for each backwards-incompatible change to the network protocols or the encoding of the consensus data types. This is clearly a strict superset of the changes caused by consensus. When the network layer introduces a new protocol message, this does not necessarily mean anything changes in the encoding of the consensus data types. This means multiple network versions can correspond to the same consensus-side encoding or *consensus version*. In the other direction, each change to the consensus-side encodings should result in a new network version. We capture this in the following abstraction:

```
class (..) => HasNetworkProtocolVersion blk where
  type BlockNodeToNodeVersion   blk :: Type
  type BlockNodeToClientVersion blk :: Type

class HasNetworkProtocolVersion blk
   => SupportedNetworkProtocolVersion blk where
  supportedNodeToNodeVersions ::
       Proxy blk -> Map NodeToNodeVersion   (BlockNodeToNodeVersion   blk)
  supportedNodeToClientVersions ::
       Proxy blk -> Map NodeToClientVersion (BlockNodeToClientVersion blk)
```

The `HasNetworkProtocolVersion` class has two associated types to define the consensus version number types for the given block. When no versioning is needed, one can use the unit type as the version number. The `SupportedNetworkProtocolVersion` defines the mapping between the network and the consensus version numbers. Note that this does not have to be an injection, as multiple network version can most certainly map to the same consensus version. Nor does this have to be a surjection, as old network and consensus versions might be removed from the mapping when the old version no longer needs to be supported. This last reason is also why this mapping is modelled with a `Map` instead of a function: it allows enumerating a subset of all defined versions, which is not possible with a function. TODO

Global numbering vs multiple block types

The `SerialiseNodeToNode` and `SerialiseNodeToClient` instances can then branch on the passed version to introduce changes to the encoding format, for example, the inclusion of the block size in the Byron header encoding.

Consider the following scenario: a change is made to one of the consensus data types, for example, a new query constructor is added the ledger query data type. This requires a new consensus and thus network version number, as older versions will not be able to decode it. What should be done when the new query constructor is sent to a node that does not support it (the negotiated version is older than the one in which the constructor was added)? If it is encoded and send, the receiving side will fail to decode it and terminate its connection. This is rather confusing to the sender, as they are left in the dark. Instead, we let the *encoder* throw an exception in this case, terminating that connection, so that the sender is at least notified of this. Ideally, we could statically prevent such cases. right? TODO

6.2.2 CBOR-in-CBOR

In section 6.3, we explain why the result of the decoder for types using *annotations* needs to be passed the original encoding as a bytestring. When reading from disk, we already have the entire bytestring in memory, so it can easily be passed to the result of the decoder. However, this is not the case when receiving a message via the network layer: the entire message, containing the annotated type(s), is decoded incrementally. When decoding CBOR, it is not possible to obtain the bytestring corresponding to what explain why right?

the decoder is decoding. To work around this, we use *CBOR-in-CBOR*: we encode the original data as CBOR and then encode the resulting bytestring as CBOR *again*. When decoding CBOR-in-CBOR, after decoding the outer CBOR layer, we have exactly the bytestring that we will need for the annotation. Next, we feed this bytestring to the original decoder, and, finally, we pass the bytestring to the function returned by the decoder.

6.2.3 Serialised

One of the duties of the consensus layer is to serve blocks and headers to other nodes in the network. link? To serve for example a block, we read it from disk, deserialise it, and then serialise it again and send it across the network. The costly deserialisation and serialisation steps cancel each other out and are thus redundant. We perform this optimisation in the following way. When reading such a block from storage, we do not read the `blk`, but the `Serialised blk`, which is a phantom type around a raw, still serialised bytestring:

```
newtype Serialised a = Serialised ByteString
```

To send this serialised block over the network, we have to encode this `Serialised blk`. As it happens, we use CBOR-in-CBOR to send both blocks and headers over the network, as described in section 6.2.2. This means the serialised block corresponds to the inner CBOR layer and that we only have to encode the bytestring again as CBOR, which is cheap.

This optimisation is only used to *send* and thus encode blocks and headers, not when *receiving* them, because each received block or header will have to be inspected and validated, and thus deserialised anyway.

As discussed in section 6.1.1, reading a header (nested in a block) from disk requires reading the context and the raw header, and then combining them before we can deserialise the header. This means the approach for serialised headers differs slightly:

```
newtype SerialisedHeader blk = SerialisedHeaderFromDepPair {
      serialisedHeaderToDepPair :: GenDepPair Serialised
                                              (NestedCtxt Header blk)
    }
```

This is similar to the `DepPair (NestedCtxt f blk)` type from section 6.1.1, but this time the raw header is wrapped in `Serialised` instead of being deserialised.

6.3 Annotations

TODO

move up? The previous two sections refer to this

The following technique is used in the Byron and Shelley ledgers for a number of data types like blocks, headers, transactions, ... The in-memory representation of, for example a block, consists of both the typical fields describing the block (header, transactions, ...), but also the *serialisation* of the block in question. The block is *annotated* with its serialisation.

The principal reason for this is that it is possible that multiple serialisations, each which a different hash, correspond to the same logical block. For example, a client sending us the block might encode a number using a binary type that is wider than necessary (e.g., encoding the number 0 using four bytes instead of a single byte). CBOR defines a *canonical format*, we call an encoding that is in CBOR's canonical format a *canonical encoding*. link?

When after deserialising a block in a non-canonical encoding, we serialise it again, we will end up with a different encoding, i.e., the canonical encoding, as we stick to the canonical format. This means the hash, which is part of the blockchain, is now different and can no longer be verified.

For this reason, when deserialising a block, the original, possibly non-canonical encoding is retained and used to annotate the block. To compute the hash of the block, one can hash the annotated serialisation.

Besides solving the issue with non-canonical encodings, this has a performance advantage, as encoding such a block is very cheap, it is just a matter of copying the in-memory annotation.

TODO

We rely on it being cheap in a few places, mention that/them?

TODO

extra memory usage

This means that the result of the decoder must be passed the original encoding as a bytestring to use as the annotation of the block or other type in question. Hence the decoder corresponding to the encoder `blk -> Encoding` has type `forall s. Decoder s (ByteString -> blk)`, which is a different instantiation of the type `a`, explaining the split of the serialisation classes used for storage (section 6.1). The original encoding is then applied to the resulting function to obtain the annotated block. This asymmetry is handled in a different way for the network serialisation, namely using CBOR-in-CBOR (section 6.2.2).

6.3.1 Slicing

TODO

discuss the slicing of annotations with an example. What is the relation between the decoded bytestring and the bytestring passed to the function the decoder returns? Talk about compositionality.

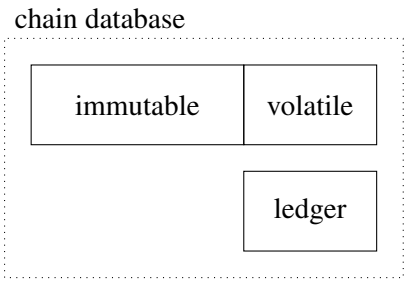
Part II

Storage Layer

Chapter 7

Overview

7.1 Components



Discuss the immutable/volatile split (we reference this section for that).

7.2 In memory

TODO: After we discussed the overview, we should give an overview of everything we store in memory in any component, so that we have a better understanding of memory usage of the chain DB as a whole.

7.2.1 Chain fragments

7.2.2 Extended ledger state

TODO: Is there a more natural place to talk about this? Introducing the header state when introducing the storage layer does not feel quite right. The storage layer might be storing the header state, but that doesn't explain its existence.

ChainDepState, (ChainIndepState), LedgerState, ExtLedgerState

Chapter 8

Immutable Database

The Immutable DB is tasked with storing the blocks that are part of the *immutable* part of the chain. Because of the nature of this task, the requirements and *non-requirements* of this component are fairly specific:

- **Append-only**: as it represents the immutable chain, blocks will only be appended in the same order as they are ordered in the chain. Blocks will never be *modified* or *deleted*.
- **Reading**: the database should be able to return the block or header stored at a given *point* (combination of slot number and hash) efficiently.
- **Efficient streaming**: when serving blocks or headers to other nodes, we need to be able to stream ranges of *consecutive* blocks or headers efficiently. As described in section 6.2.3, it should be possible to stream *raw* blocks and headers, without serialising them.
- **Recoverability**: it must be possible to validate the blocks stored in the database. When a block in the database is corrupt or missing, it is sufficient to truncate the database, representing an immutable chain, to the last valid block before the corrupt or missing block. The truncated blocks can simply be downloaded again. It is therefore not necessary to be able recover the full database when blocks are missing.

define point somewhere?

While we touched upon some of the non-requirements already above, it is useful to highlight the following non-requirements.

- **Queries**: besides looking up a single block by its point and streaming ranges of consecutive blocks, the database does have to be able to answer queries about blocks. No searching or filtering is needed.
- **Durability**: the system does not require the durability guarantee that traditional database systems provide (the D in ACID). If the system crashes right after appending a block, it is acceptable that the block in question is truncated when recovering the database. Because of the overlap with the Volatile DB, such a truncation is likely to even go unnoticed. In the worst case, the truncated blocks can simply be downloaded again.

link

Because of the specific requirements and non-requirements listed above, we decided to write our own implementation, the `ImmutableDB`, instead of using an existing off-the-shelf database system. Traditional database systems provide guarantees that are not needed and, conversely, do not take advantage of the requirements to optimise certain operations. For example, there is no need for a journal or flushing (`fsync`) the buffers after each write because of our unique durability and recoverability (non-)requirements.

8.1 API

Before we describe the implementation of the Immutable DB, we first describe its functionality. The Immutable DB has the following API:

```
data ImmutableDB m blk = ImmutableDB {
      closeDB :: m ()

    , getTip :: STM m (WithOrigin (Tip blk))

    , appendBlock :: blk -> m ()

    , getBlockComponent ::
          forall b.
          BlockComponent blk b
       -> RealPoint blk
       -> m (Either (MissingBlock blk) b)

    , stream ::
          forall b.
          ResourceRegistry m
       -> BlockComponent blk b
       -> StreamFrom blk
       -> StreamTo   blk
       -> m (Either (MissingBlock blk) (Iterator m blk b))
    }
```

The database is parameterised over the block type `blk` and the monad `m`, like most of the consensus layer. Mention our use of records for components?

mention io-sim

The `closeDB` operation closes the database, allowing all opened resources, including open file handles, to be released. This is typically only used when shutting down the entire system. Calling any other operation on an already-closed database should result in an exception.

TODO

The `getTip` operation returns the current tip of the Immutable DB. The `Tip` type contains information about the block at the tip like the slot number, the block number, the hash, etc. The `WithOrigin` type is isomorphic to **`Maybe`** and is used to account for the possibility of an empty database, i.e., when the tip is at the "origin" of the chain. This operation is an `STM` operation, which allows it to be combined with other `STM` operations in a single transaction, to obtain a consistent view on them. This also implies that no IO or disk access is needed to obtain the current tip.

The `appendBlock` operation appends a block to the Immutable DB. As slot numbers increase monotonically in the blockchain, the block's slot must be greater than the current tip's slot (or equal when the tip points at an EBB, see chapter 23). It is not required that each slot is filled, so there can certainly be gaps in the slot numbers.

link?

The `getBlockComponent` operation allows reading one or more components of the block in the database at the given point. We discuss what block components are in section 8.1.1. The `RealPoint` type represents a point that can only refer to a block, not to genesis (the empty chain), which the larger `Point` type allows. As the given point might not be in the Immutable DB, this operation can also return a `MissingBlock` error instead of the requested block component.

The `stream` operation returns an iterator to efficiently stream the blocks between the two given bounds. The bounds are defined as such:

```
data StreamFrom blk =
    StreamFromInclusive !(RealPoint blk)
  | StreamFromExclusive !(Point     blk)

newtype StreamTo blk =
    StreamToInclusive (RealPoint blk)
```

Lower bounds can be either inclusive or exclusive, but exclusive upper bounds were omitted because they were not needed in practice. An inclusive bound must refer to a block, not genesis, hence the use of `RealPoint`. The exclusive lower bound *can* refer to genesis, hence the use of `Point`, in particular to begin streaming from the start of the chain. As one or both of the bounds might not be in the Immutable DB, this operation can return a `MissingBlock` error. We discuss what block components are in section 8.1.1. The `ResourceRegistry` will be used to allocate all the resources the iterator opens during its lifetime, e.g., file handles. By closing the registry in case of an exception (using **`bracket`**), all open resources are released and nothing is leaked. More discussion about iterators follows in section 8.1.2.

link

8.1.1 Block Component

TODO

move to ChainDB?

Besides reading or streaming blocks from the Immutable DB, it must be possible to read or stream headers, raw blocks (see section 6.2.3), but in some cases also *nested contexts* (see section 6.1.1) or even block sizes. Adding an operation to the API for each of these would result in too much duplication. We handle this with the `BlockComponent` abstraction: when reading or streaming, one can choose which *components* of a block should be returned, e.g., the block itself, the header of the block, the size of the block, the raw block, the raw header, etc. We model this with the following GADT:

```
data BlockComponent blk a where
  GetVerifiedBlock :: BlockComponent blk blk
  GetBlock         :: BlockComponent blk blk
  GetRawBlock      :: BlockComponent blk ByteString
  GetHeader        :: BlockComponent blk (Header blk)
  GetRawHeader     :: BlockComponent blk ByteString
  GetHash          :: BlockComponent blk (HeaderHash blk)
  GetSlot          :: BlockComponent blk SlotNo
  GetIsEBB         :: BlockComponent blk IsEBB
  GetBlockSize     :: BlockComponent blk Word32
  GetHeaderSize    :: BlockComponent blk Word16
  GetNestedCtxt    :: BlockComponent blk (SomeSecond (NestedCtxt Header) blk)
  ..
```

The `a` type index determines the type of the block component. Additionally, we have **`Functor`** and `Applicative` instances. The latter allows combining multiple `BlockComponents` into one. This can be considered a small DSL for querying components of a block.

8.1.2 Iterators

The following API can be used to interact with an iterator:

```
data Iterator m blk b = Iterator {
      iteratorNext    :: m (IteratorResult b)
    , iteratorHasNext :: STM m (Maybe (RealPoint blk))
    , iteratorClose   :: m ()
    }

data IteratorResult b =
    IteratorExhausted
  | IteratorResult b
```

The `iteratorNext` operation returns the current `IteratorResult` and advances the iterator to the next block in the stream. When the iterator has reached its upper bound, it is exhausted. Remember that the `b` type argument corresponds to the requested block component.

The `iteratorHasNext` operation returns the point corresponding to the block the next call to `iteratorNext` will return. When exhausted, **`Nothing`** is returned.

As an open iterator can hold onto resources, e.g., open file handles, it should be explicitly closed using the `iteratorClose` operation. Interacting with a closed iterator should result in an exception, except for calling `iteratorClose`, which is idempotent.

8.2 Implementation

We will now give a high-level overview of our custom implementation of the Immutable DB that satisfies the requirements and the API.

- We store blocks sequentially in a file, called a *chunk file*. We append each raw block, without any extra information before or after it, to the chunk file. This will facilitate efficient binary streaming of blocks. In principle, it is a matter of copying bytes from one buffer to another, without any additional processing needed.

- Every *x slots*, where *x* is the configurable chunk size, we start a new chunk file to avoid storing all blocks in a single file.

- To facilitate looking up a block by a point, which consists of the hash and the slot number, we "index" our database by slot numbers. One can then look up the block in the given slot and compare its hash against the point's hash. No searching will be needed.

  Blocks are stored sequentially in chunk files, but slot numbers do *not* increase one-by-one; they are *sparse*. This means we need a mapping from the slot number to the offset and size of the block in the chunk file. We store this mapping in the on-disk *primary index*, one per chunk file, which we discuss in more detail in section 8.2.2.

- As mentioned above, when looking up a block by a point, we will compare the hash of the block at the point's slot in the Immutable DB with the point's hash. We should be able to do this without first having to read and deserialise the entire block in order to know its hash.

  Moreover, it should be possible to read just the header of the block without first having to read the entire block. As described in section 6.1.1, we can do this if have access to the header offset, header size, and nested context of the block.

  For these reasons, we store the aforementioned extra information, which should be available without having to read and deserialise the entire block, separately in the on-disk *secondary index*, one per chunk file (section 8.2.2).

- All the information stored in the primary and secondary indices can be recovered from the blocks in the chunk files. This is described in section 8.2.3.

- Whenever a file-system operation fails, or a file is missing or corrupted, we shut down the Immutable DB and consequently the whole system. When this happens, either the system's file system is no longer reliable (e.g., disk corruption), manual intervention (e.g., disk is full) is required, or there is a bug in the system. In all cases, there is no point in trying to continue operating. We shut down the system and flag the shutdown as *dirty*, triggering a full validation on the next start-up, see section 8.2.3.

  Not all forms of disk corruption can easily be detected. For example, when some bytes in a block stored in a chunk file have been flipped on disk, this can easily go unnoticed. Deserialising the block might fail if the serialisation format is no longer valid, but the bitflip could also happen in, e.g., the amount of a transaction, which will not be detected by the deserialiser. In fact, the majority of blocks read will not even be deserialised, as blocks served to other nodes are read and sent in their raw, still serialised format. However, sending a corrupted block must be avoided, as nodes receiving it will consider it invalid and can blacklist us, mistaking us for an adversary.

To detect such forms of silent corruption, we store CRC32 checksums in the secondary index (section 8.2.2) which we verify when reading the block, which we can do even when not deserialising the block. Note that we could use the block's own hash for this purpose,[19] but because computing such a cryptographic hash is much more expensive, we opted for a separate CRC32 checksum, which is much more efficient to compute and designed for exactly this purpose.

- We store the state of the current chunk, including its indices, in memory. We store this state, a pure data type, in a `StrictMVar`. Besides avoiding space leaks by forcing its contents to WHNF, this `StrictMVar` type has another useful ability that its standard non-strict variant is lacks: while it is locked when being modified, the previous, *stale* value can still be read.

  This is convenient for the Immutable DB: we can support multiple concurrent reads even when at most one append operation is in progress, as it is safe to read a block based on the stale state because data will only be appended, not modified.

  To append a block to the Immutable DB, we lock the state to avoid concurrent append operations. We append the block to the chunk file, and append the necessary information to the primary and secondary indices. We unlock the state, updated with the information about the newly appended block.

- We *do not flush* any writes to disk, as discussed in the introduction of this chapter. This makes appending a block quite cheap: the serialised block is copied to an OS buffer, which is then asynchronously flushed in the background.

- To avoid repeatedly reading and deserialising the same primary and secondary indices of older chunks, we cache them in a LRU-cache that is bounded in size.

- To open an iterator, we check its bounds using the (cached) indices. The bounds are valid when both correspond to blocks present in the Immutable DB. Next, a file handle is opened for the chunk file containing the first block to stream. The same chunk file's indices are read (from the cache) and the iterator will maintain a list of secondary index entries, one for each block to stream from the chunk file. By having this list of entries in memory, the indices will not have to be accessed for each streamed block.

  When a block component is requested from the iterator, it is read from the chunk file and/or extracted from the corresponding in-memory entry. Afterwards, the entry is dropped from the in-memory list so that the next entry is ready to be read. When the list of entries is exhausted without reaching the end bound, we move on to the next chunk file. This process repeats until the end bound is reached.

8.2.1 Chunk layout

Each block in the block chain has a unique slot number (except for EBBs, which we discuss below). Slot numbers increase in the blockchain, but not all slots have to be filled. For example, in the Byron era (using the Permissive BFT consensus algorithm), nearly every slot will be filled, but in the Shelley era (using the Praos consensus algorithm), on average only one in twenty slots will be filled.

As mentioned above, we want to group blocks into chunk files. Because we need to be able to look up blocks in the Immutable DB based on their slot number, we group blocks into chunk files based on their slot numbers so that the chunk file containing a block can be determined by looking at the slot number of the block.

Internally, we translate *absolute* slot numbers into *chunk numbers* and *relative slot numbers* (relative w.r.t. the chunk). As EBBs (chapter 23) have the same slot number as their successor, this translation is not injective. To restore injectivity, we include "whether the block is an EBB or not" as an input to the translation. TODO

[19] To be precise: we would have to check the block body against the body hash stored in the header, and verify the signature of the header.

how should this be formatted?

Definition 8.1 (Chunk number)**.** Let s be the absolute slot number of a block. Using a chunk size of sz:

$$\mathsf{chunkNumber}(s) = \lfloor s/sz \rfloor$$

Naturally, chunks are zero-indexed.

Definition 8.2 (Relative slot number)**.** Let s be the absolute slot number of a block. Using a chunk size of sz:

$$\mathsf{relativeSlot}(s, isEBB) = \begin{cases} 0 & \text{if } isEBB \\ (s \bmod sz) + 1 & \text{otherwise} \end{cases}$$

We reserve the very first relative slot for an EBB, hence the need to make room for it by incrementing by one in the non-EBB case.

In the example below, we show a chunk with chunk number 1 using a chunk size of 100:

	EBB	Block	Block	Block	⋯	Block	Block
Absolute slot numbers	100	100	101	103		197	199
Relative slot numbers	0	1	2	4		98	100

Note that some slots are empty, e.g., 102 and 198 are missing. The first and lasts slots can be empty too. In practice, it will never be the case that an entire chunk is empty, but the implementation allows for it.

If we were to pick a chunk size of 1 and store each block in its own file, we would need millions of files, as there are millions of blocks. When serving blocks to peer, we would constantly open and close individual block files, which is very inefficient.

If we pick a very large or even unbounded chunk size, the resulting chunk file would be several gigabytes in size and keep growing. This would make the recovery process (section 8.2.3) more complicated and potentially much slower, as more data might have to be read and validated. Moreover, our current approach of caching indices per chunk would have to be revised.

Other arguments?

In practice, a chunk size of 21,600 is used, which matches the *epoch size* of Byron. It is no coincidence that there is (at most) one EBB at the start of each Byron epoch, fitting nicely in the first relative slot that we reserve for it. Originally, the Immutable DB called these chunk files *epoch files*. With the advent of Shelley, which has a different epoch size than Byron, we decoupled the two and introduced the name "chunk".

Dynamic chunk size The *chunking* scheme was designed with the possibility of a non-uniform chunk size in mind. Originally, the goal was to make the chunk size configurable such that the number of slots per chunk could change after a certain slot. Similarly, the reserving an extra slot for an EBB would be optional and could stop after a certain slot, i.e., when the production of EBBs stopped. The reasoning behind this was to allow the chunk size to change near the transition from Byron to Shelley. As the slot density goes down by a factor of twenty when transitioning to Shelley, the number of blocks per chunk file and, consequently, the chunk size would go down by the same factor, leading to too many, smaller chunk files. The intention was to configure the chunk size to increase by the same factor at the start of the Shelley era.

The transition to another era, e.g., Shelley, is dynamic: the slot at which it happens is determined by on-chain voting and is not certain up to a number of hours in advance. Making the mapping from slot number to chunk and relative slot number rely on the actual slot at which the transition happened would complicate things significantly. It would make the mapping depend on the ledger state, which determines

the current era. This would make an unwanted coupling between the Immutable DB, storing *blocks*, to the ledger state obtained by applying these blocks. A reasonable compromise would be to hard-code the change in chunk size to the estimated transition slot. When the estimate is incorrect, only a few Byron chunks would contain more blocks than intended or only a few Shelley chunks would contain fewer blocks than intended.

Unfortunately, due to lack of time, dynamic chunk sizes were not implemented in time for the transition to Shelley. This means the same chunk size is being used for the *entire chain*, resulting in fewer blocks per Shelley chunk file than ideal, and, consequently, more chunk files than ideal.

Indexing by block number The problem of too small, too many chunk files described in the paragraph above is caused by the fact that slot numbers can be sparse and do not have to be consecutive. *Block numbers* do not have the same problem: they are consecutive and thus dense, regardless the era of the blockchain. If instead of indexing the Immutable DB by slot numbers, we indexed it by *block numbers*, we would not have the problem. Unfortunately, the point type, which is used throughout the network layer and the consensus layer to identify and look up blocks, consists of a hash and *slot number*, not a block number.

Either we would have to maintain another index from slot number to block number, which would require its own chunking scheme based on slot numbers or just one big file, which has its own downsides. Or, points should be based on block numbers instead of slot numbers. As points are omnipresent, this change is very far-reaching. The latter approach carries our preference, but is currently out of the question. The former is more localised, but the complexity and risks involved in migrating deployed on-disk databases to the new format do not outweigh the uncertain benefits.

8.2.2 Indices

As mentioned before, we have on-disk indices for the chunk files for two purposes:

1. To map the sparse slot numbers to the blocks that are densely stored in the chunk files.
2. To store the information about a block that should be available without having to read and deserialise the actual block, e.g., the header offset, the header size, the CRC32 checksum, etc.

We use a separate index for each task: the *primary index* for the first task and the *secondary index* for the second task. Each chunk file has a corresponding primary index file and secondary index file. Because of a dependency of the primary index on the secondary index, we first discuss the latter.

Secondary index In the secondary index, we store the information about a block that is needed before or without having to read and deserialise the block. The secondary index is an append-only file, like the chunk file, and contains a *secondary index entry* for each block. For simplicity and robustness, a secondary index merely contains a series of densely stored secondary index entries with no extra information between, before, or after them. This avoids needing to initialise or finalise such a file, which also makes the recovery process simpler (section 8.2.3). A secondary index entry consists of the following fields:

field	size [bytes]
block offset	8
header offset	2
header size	2
checksum	4
header hash	X
block or EBB	8

- The block offset is used to determine at which offset in the corresponding chunk file the raw block can be read.

As blocks are variable-sized, the size of the block also needs to be known in order to read it. Instead of spending another 8 bytes to store the block size as an additional field, we read the block offset of the *next entry* in the secondary index, which corresponds to the block after it. The block size can then be computed by subtracting the latter's block offset from the former's.

In case the block is the final block in the chunk file, there is no next entry. Instead, the final block's size can be derived from the chunk file's size. When reading the final block B_n of the current chunk file, it is important to obtain the chunk file size at the right time, before any more blocks ($B_{n+1}, B_{n+2}, \ldots$) are appended to the same file, increasing the chunk file size. Otherwise, we risk reading the bytes corresponding to not just the previously final block B_n, but also $B_{n+1}, B_{n+2}, \ldots$.[20]

The reasoning behind using 8 bytes for the block offset is the following. The maximum block header and block body sizes permitted by the blockchain itself are dynamic parameters that can change through on-chain voting. At the time of writing, the maximum header size is 1100 bytes and the maximum body size is 65,536 bytes. By multiplying this theoretical maximum block size of 1100 + 65,536 = 66,636 bytes by the chunk size used in practice, i.e., 21,600, assuming a maximal density of 1.0 in the Byron era, we get 1,439,337,600 as the maximal file size for a chunk file. An offset into a file of that size fits tightly in 4 bytes, but this would not support any future maximum block size increases, hence the decision to use 8 bytes.

- The header offset and header size are needed to extract the header from a block without first having to read and deserialise the entire block, as discussed in section 6.1.1. These are stored per block, as the header size can differ from block to block. The nested context is reconstructed by reading bytes from the start of the block, as explained in our discussion of the `ReconstructNestedCtxt` class in section 6.1.1.

  Using 2 bytes for the header offset and header size is enough when taking the following into account: (so far all types of) blocks start with their header, the current maximum header size is 1100 bytes, and the header offset is relative to the start of the block.

- As discussed before, to detect silent corruption of blocks, we store a CRC checksum of each block, against which the block is verified after reading it. This verification can be done without deserialising the block.

  Note that we do not store a checksum of the raw header, which means we do not check for silent corruption when streaming headers.

  Maybe we should?

- The header hash is used for lookups and bounds checking, i.e., to check whether the given point's hash matches the hash of the block as the same slot in the Immutable DB. By storing it separately, we do not have to read and compute the hash of the block's header just to check whether it has the right hash.

  The header hash field's size depends on the concrete instantiation of the `HeaderHash blk` type family. In practice, a 32-byte hash is used.

- The "block or EBB" field is represented in memory as follows:

```
data BlockOrEBB =
    Block !SlotNo
  | EBB   !EpochNo
```

  The former constructor represents a regular block with an absolute slot number and the latter an EBB (chapter 23) with an epoch number (since there is only a single EBB per epoch). The main reason this field is part of the secondary index entry is to implement the `iteratorHasNext` method of the iterator API (see section 8.1.2) without having to read the next block from disk, as the iterator will keep these secondary index entries in memory.

[20] In hindsight, storing the block size as a separate field would have simplified the implementation.

Both the `SlotNo` and `EpochNo` types are newtypes around a `Word64`, hence the 8 on-disk bytes. We omit the tag distinguishing between the two constructors in the serialisation because in nearly all cases, this information has already been retrieved from the primary index, i.e., whether the first filled slot in a chunk is an EBB or not.[21]

- Because of the fixed size of each field, it was originally decided to (de)serialise the corresponding data type using the `Binary` class. Using CBOR would be more flexible to future changes. This would make the encoding variable-sized, which is not necessarily an issue, which will become clear in our description of the primary index.

Primary index The primary index maps the sparse slot numbers to the secondary index entries of the corresponding blocks in the dense secondary index. As discussed above, the secondary index entry of a block tells us the offset in the chunk file of the corresponding block.

The format of the primary index is as follows. The primary index start with a byte indicating its version number. Next, for each slot, empty or not, we store the offset at which its secondary index entry starts in the secondary index. This same offset will correspond to the *end* of the previous secondary index entry. When a slot is empty, its offset will be the same as the offset of the slot before it, indicating that the corresponding secondary index entry is empty.

When appending a new block, we append the previous offset as many times as the number of slots that was skipped, indicating that they are empty. Next, we append the offset after the newly appended secondary index entry corresponding to the new block.

We use a fixed size of 4 bytes to store each offset. As this is an offset in the secondary index, it should be at least large enough to address the maximal size of a secondary index file. We can compute this by multiplying the used chunk size by the size of a secondary index entry: $21{,}600 * (8 + 2 + 2 + 4 + 32 + 8) = 1{,}209{,}600$, which requires more than 2 bytes to address.

To look up the secondary index entry for a certain slot, we compute the corresponding chunk number and relative slot number using chunkNumber and relativeSlot (we discuss how we deal with EBBs later). Because we use a fixed size for each offset, based on the relative slot number, we can compute exactly at which bytes should be read at which offset in the primary index, i.e., the 4 + 4 bytes corresponding to the offset at the relative slot and the offset after it. When both offsets are equal, the slot is empty. When not equal, we now know which bytes to read from the secondary index to obtain the secondary index entry corresponding to the block in question.

However, as mentioned in section 8.2, we maintain a cache of primary indices, which means that they are always read from disk in their entirety. After a cache hit, looking up a relative slot in the cached primary index corresponds to a constant-time lookup in a vector.

We illustrate this format with an example primary index below, which matches the chunk out of the example from section 8.2.1. The offsets correspond to the blocks on the line below them, where ∅ indicates an empty slot. We assume a fixed size of 10 bytes for each secondary index entry. The offset X corresponds the final size of the secondary index.

Offsets	0	10	20	30	30	40	⋯	$X-20$	$X-10$	$X-10$	X
Blocks	EBB	Block	Block	∅	Block		⋯	Block	∅	Block	
Absolute slot numbers	100	100	101	102	103		⋯	197	198	199	
Relative slot numbers	0	1	2	3	4		⋯	98	99	100	

The version number we mentioned above can be used to migrate indices in the old format to a newer format, when the need would arise in the future. We do not include a version number in the secondary

[21] In hindsight, having the tag in the serialisation would have simplified the implementation.

index, as both index formats are tightly coupled, which means that both index files should be migrated together.

One might realise that because the size of a secondary index entry is static, the primary index could be represented more compactly using a bitmap. This is indeed the case and the reason for it not being a bitmap is mostly a historical accident. However, this accident has the upside that migrating to variable-sized secondary index entries, e.g., serialised using CBOR instead of `Binary` is straightforward.

Lookup Having discussed both index formats, we can now finally detail the process of looking up a block by a point. Given a point with slot s and hash h, we need to go through the following steps to read the corresponding block:

1. Determine the chunk number $c = \mathsf{chunkNumber}(s)$.

2. Determine the relative slot rs within chunk c corresponding to s: $rs = \mathsf{relativeSlot}(s, isEBB)$.

   Note the *isEBB* argument, which is unknown at this point. Just by looking at the slot and the static chunk size, we can tell whether the block *could* be an EBB or not: only the very first slot in a chunk (which has the same size as a Byron epoch) could correspond to an EBB *or* the regular block after it. For all other slots we are certain they cannot correspond to an EBB.

   In case the slot s corresponds to the very first slot in the chunk, we will have to use the hash h to determine whether the point corresponds to the EBB or the regular block in slot s.

3. We lookup the offset at rs and the offset after it in the primary index of chunk c. As discussed, these lookups go through a cache and are cheap. We now have the offsets in the secondary index file corresponding to the start and end of the secondary index entry we are interested in. If both offsets are equal, the slot is empty, and the lookup process terminates.

   In case of a potential EBB, we have to do two such lookups: once for relative slot 0 and once for relative slot 1.

4. We read the secondary index entry from the secondary index file. The secondary indices are also cached on a per chunk basis. The secondary index entry contains the header hash, which we can now compare against h. In case of a match, we can read the block from the chunk file using the block offset contained in the secondary index entry. When the hash does not match, the lookup process terminates.

   In case of a potential EBB, the hash comparisons finally tell us whether the point corresponds to the EBB or the regular block in slot s, or *neither* in case both hashes do not match h.

8.2.3 Recovery

Because of the specific requirements of the Immutable DB and the expected write patterns, we can use a much simpler recovery scheme than traditional database systems. Only the immutable, append-only part of the chain is stored, which means that data inconsistencies (e.g., because of a hard shutdown) are most likely to happen at the end of the chain, i.e., in the last chunk and its indices. We can simply truncate the chain in such cases. As we maintain some overlap with the Volatile DB, blocks truncated from the end of the chain are likely to still be in the Volatile DB, making the recovery process unnoticeable. If the overlap is not enough and the truncated blocks are not in the Volatile DB, they can simply be downloaded again.

link

There are two modes of recovery:

1. Validate the last chunk: this is the default mode when opening the Immutable DB. The last chunk file and its indices are validated. This will detect and truncate append operations that did not go through entirely, e.g., a block that was only partially appended to a chunk file, or a block that was appended to one or both of the indices, but not to the chunk file.

When after truncating a chunk file, the chunk file becomes empty, we validate the chunk file before it. In the unlikely case that that chunk file has to be truncated and ends up empty too, we validate the chunk file before it and so on, until we reach a valid block or the database is empty.

2. Validate all chunks: this is the full recovery mode that is triggered by a dirty shut down, caused by a missing or corrupted file (e.g., a checksum mismatch while reading), or because the node itself was not shut down properly. We validate all chunk files and their indices, from oldest to newest. When a corrupt or missing block is encountered, we truncate the chain to the last valid block before it. Trying to recover from a chain with holes in it would be terribly complex, we therefore do not even try it.

In the latter case, validating the last would be enough

In both recovery modes, chunks are validated the same way, which we will shortly describe. When in full recovery mode, we also check whether the last block in a chunk is the predecessor of the first block in the next chunk, by comparing the hashes. This helps sniff out a truncated chunk file that is not the final one, causing a gap in the chain.

Validating a chunk proceeds as follows:

- In the common case, the chunk file and the corresponding primary and secondary index files will be present and all valid. We optimise for this case.[22]
- The secondary index contains a CRC32 checksum of each block in the corresponding chunk (see section 8.2.2), we extract these checksums and pass them to the *chunk file parser*.
- The chunk file parser will try to deserialise all blocks in a chunk file. When a block fails to deserialise, it is treated as corrupt and we truncate the chain to the last valid block before it. Each raw block is also checked against the CRC32 checksum from the secondary index, to detect corruptions that are not caught by deserialising, e.g., flipping a bit in a `Word64`, which can remain a valid, yet corrupt `Word64`.[23]

  When the CRC32 checksum is not available, because of a missing or partial secondary index file, we fall back to the more expensive validation of the block based on its cryptographic hashes to detect silent corruption. This type of validation is block-dependent and provided in the form of the `nodeCheckIntegrity` method of the `NodeInitStorage` class. This validation is implemented by hashing the body of the block and comparing it against the body hash stored in the header, and by verifying the signature of the header.

  When the CRC32 checksum *is* available, but does not match the one computed from the raw block, we also fall back to this validation, as we do not know whether the checksum or the block was corrupted (although the latter is far more likely).

  The chunk file parser also verifies that the hashes line up within a chunk, to detect missing blocks. It does this by comparing the "previous hash" of each block with the previous block's hash.

  The chunk file parser returns a list of secondary index entries, forming together the corresponding secondary index.

- The chunk file containing the blocks is our source of truth. To check the validity of the secondary index, we check whether it matches the secondary index returned by the chunk file parser. If there is a mismatch, we overwrite the entire secondary index file using the secondary index returned by the chunk file parser.
- We can reconstruct the primary index from the secondary index returned by the chunk file parser. When the on-disk primary index is missing or it is not equal to the reconstructed one, we (over)write it using the reconstructed one.

[22] Unlike in other areas, where we try to maintain that the average case is equal to the worst case.

[23] One might think that deserialising the blocks is not necessary if the checksums all match. However, the chunk file parser also the corresponding secondary index, which is used to validate the on-disk one. For this process deserialisation is required.

- When truncating the chain, we always make sure that the resulting chain ends with a block, i.e., a filled slot, not an empty slot. Even if this means going back to the previous chunk.

We test the recovery process in our `quickcheck-state-machine` tests of the Immutable DB. In various states of the Immutable DB, we generate one or more random corruptions for any of the on-disk files: either a simple deletion of the file, a truncation, or a random bitflip. We verify that after restarting the Immutable DB, it has recovered to the last valid block before the truncation.

Additionally, in those same tests, we simulate file-system errors during operations. For example, while appending a block, we let the second disk write fail. This is another way of testing whether we can correctly recover from a write that was aborted half-way through.

Chapter 9

Volatile Database

The Volatile DB is tasked with storing the blocks that are part of the *volatile* part of the chain. Do not be misled by its name, the Volatile DB *should persist* blocks stored to disk. The volatile part of the chain consists of the last k (the security parameter, see section 4.1.2) blocks of the chain, which can still be rolled back when switching to a fork. This means that unlike the Immutable DB, which stores the immutable prefix of *the* chosen chain, the Volatile DB can store potentially multiple chains, one of which will be the current chain. It will also store forks that we have switched away from, or will still switch to, when they grow longer and become preferable to our current chain. Moreover, the Volatile DB can contain disconnected blocks, as the block fetch client might download or receive blocks out of order. link

We list the requirements and non-requirements of this component in no particular order. Note that some of these requirements were defined in response to the requirements of the Immutable DB (see chapter 8), and vice versa.

- **Add-only**: new blocks are always added, never modified.
- **Out-of-order**: new blocks can be added in any order, i.e., consecutive blocks on a chain are not necessarily added consecutively. They can arrive in any order and can be interspersed with blocks from other chains.
- **Garbage-collected**: blocks in the current chain that become older than k, i.e., there are at least k more recent blocks in the current chain after them, are copied from the Volatile DB to the Immutable DB, as they move from the volatile to the immutable part of the chain. After copying them to the Immutable DB, they can be *garbage collected* from the Volatile DB.

  Blocks that are not part of the chain but are too old to switch to, should also be garbage collected.
- **Overlap**: by allowing an *overlap* of blocks between the Immutable DB and the Volatile DB, i.e., by delaying garbage collection so that it does not happen right after copying the block to the Immutable DB, we can weaken the durability requirement on the Immutable DB. Blocks truncated from the end of the Immutable DB will likely still be in the Volatile DB, and can simply be copied again. done by ChainDB
- **Durability**: similar to the Immutable DB's durability *non-requirement*, losing a block because of a crash in the middle or right after appending a block is inconsequential. The block can be downloaded again.
- **Size**: because of garbage collection, there is a bound on the size of the Volatile DB in terms of blocks: in the order of k, which is 2160 for mainnet (we give a more detailed estimate of the size in section 9.2.1). This makes the size of the Volatile DB relatively small, allowing for some information to be kept in memory instead of on disk.
- **Reading**: the database should be able to return the block or header corresponding to the given hash efficiently. Unlike the Immutable DB, we do not index by slot numbers, as multiple blocks, from different forks, can have the same slot number. Instead, we use header hashes.

- **Queries**: it should be possible to query information about blocks. For example, we need to be able to efficiently tell which blocks are stored in the Volatile DB, or construct a path through the Volatile DB connecting a block to another one by chasing its predecessors.[24] Such operations should produce consistent results, even while blocks are being added and garbage collected concurrently.
- **Recoverability**: because of its small size and it being acceptable to download missing blocks again, it is not of paramount importance to be able to recover as many blocks as possible in case of a corruption.

  However, corrupted blocks should be detected and deleted from the Volatile DB.
- **Efficient streaming**: while blocks will be streamed from the Volatile DB, this requirement is not as important as it is for the Immutable DB. Only a small number of blocks will reside in the Volatile DB, hence fewer blocks will be streamed. Most commonly, the block at the tip of the chain will be streamed from the Volatile DB (and possibly some of its predecessors). In this case, efficiently being able to read a single block will suffice.

9.1 API

Before we describe the implementation of the Volatile DB, we first describe its functionality. The Volatile DB has the following API:

```
data VolatileDB m blk = VolatileDB {
      closeDB :: m ()

    , putBlock :: blk -> m ()

    , getBlockComponent ::
          forall b.
          BlockComponent blk b
      -> HeaderHash blk
      -> m (Maybe b)

    , garbageCollect :: SlotNo -> m ()

    , getBlockInfo :: STM m (HeaderHash blk -> Maybe (BlockInfo blk))

    , filterByPredecessor :: STM m (ChainHash blk -> Set (HeaderHash blk))

    , getMaxSlotNo :: STM m MaxSlotNo
    }
```

The database is parameterised over the block type `blk` and the monad `m`, like most of the consensus layer. Mention our use of records for components?

mention io-sim

The `closeDB` operation closes the database, allowing all opened resources, including open file handles, to be released. This is typically only used when shutting down the entire system. Calling any other operation on an already-closed database should result in an exception.

TODO

The `putBlock` operation adds a block to the Volatile DB. There are no requirements on this block. This operation is idempotent, as duplicate blocks are ignored.

The `getBlockComponent` operation allows reading one or more components of the block in the database with the given hash. See section 8.1.1 for a discussion about block components. As no block with the given hash might be in the Volatile DB, this operation returns a **`Maybe`**.

The `garbageCollect` operation will try to garbage collect all blocks with a slot number less than the given one. This will be called after copying a block with the given slot number to the Immutable DB.

[24] Note that implementing this efficiently using SQL is not straightforward.

Note that the condition is "less than", not "less than or equal to", even though after a block with slot *s* has become immutable, any other blocks produced in the same slot *s* can never be adopted again and can thus safely be garbage collected. Moreover, the block we have just copied to the Immutable DB will not even be garbage collected from the Volatile DB (that will be done after copying its successor and triggering a garbage collection for the successor's slot number).

The reason for "less than" is because of EBBs (chapter 23). An EBB has the same slot number as its successor. This means that if an EBB has become immutable, and we were to garbage collected all blocks with a slot less than or *equal* to its slot number, we would garbage collect its successor block too, before having copied it to the Immutable DB.

The next two operations, `getBlockInfo` and `filterByPredecessor`, allow querying the Volatile DB. Both operations are `STM`-transactions that return a function. This means that they can both be called in the same transaction to ensure they produce results that are consistent w.r.t. each other.

The `getBlockInfo` operation returns a function to look up the `BlockInfo` corresponding to a block's hash. The `BlockInfo` data type is defined as follows:

```
data BlockInfo blk = BlockInfo {
      biHash          :: !(HeaderHash blk)
    , biSlotNo        :: !SlotNo
    , biBlockNo       :: !BlockNo
    , biPrevHash      :: !(ChainHash blk)
    , biIsEBB         :: !IsEBB
    , biHeaderOffset  :: !Word16
    , biHeaderSize    :: !Word16
    }
```

This is similar to the information stored in the Immutable DB's on-disk indices, see section 8.2.2. However, in this case, the information has to be retrieved from an in-memory index, as the function returned from the `STM` transaction is pure.

The `filterByPredecessor` operation returns a function to look up the successors of a given `ChainHash`. The `ChainHash` data type is defined as follows:

Explain somewhere else and link?

```
data ChainHash b =
    GenesisHash
  | BlockHash !(HeaderHash b)
```

This extends the header hash type with a case for genesis, which is needed to look up the blocks that fit onto genesis. As the Volatile DB can store multiple forks, multiple blocks can have the same predecessor, hence a *set* of header hashes is returned. This mapping is derived from the "previous hash" stored in each block's header. Consequently, the set will only contain the header hashes of blocks that are currently in the Volatile DB. Hence the choice for the `filterByPredecessor` name instead of the slightly misleading `getSuccessors`. This operation can be used to efficiently construct a path between two blocks in the Volatile DB. Note that only a single access to the Volatile DB is need to retrieve the function instead of an access *per lookup*.

The final operation, `getMaxSlotNo`, is also an STM query, returning the highest slot number stored in the Volatile DB so far. The `MaxSlotNo` data type is defined as follows:

```
data MaxSlotNo =
    NoMaxSlotNo
  | MaxSlotNo !SlotNo
```

This is used as an optimisation of fragment filtering in the block fetch client, look up the `filterWithMaxSlotN` link function for more information.

9.2 Implementation

We will now give a high-level overview of our custom implementation of the Volatile DB that satisfies the requirements and the API.

- We append each new block, without any extra information before or after it, to a file. When x blocks have been appended to the file, the file is closed and a new file is created.

  The smaller x, the more files are created. The higher x, the longer it will take for a block to be garbage collected, as explained in section 9.2.1. The default value for x is currently 1000.

  mention downsides

  For each file, we track the following information:

  ```
  data FileInfo blk = FileInfo {
      maxSlotNo :: !MaxSlotNo
    , hashes    :: !(Set (HeaderHash blk))
    }
  ```

  The `maxSlotNo` field caches the highest slot number stored in the file. To compute the global `MaxSlotNo`, we simply take the maximum of these `maxSlotNo` fields.

- We *do not flush* any writes to disk, as discussed in the introduction of this chapter. This makes writing a block quite cheap: the serialised block is copied to an OS buffer, which is then asynchronously flushed in the background.

- Besides tracking some information per file, we also maintain two in-memory indices to implement the `getBlockInfo` and `filterByPredecessor` operations.

  The first index, called the `ReverseIndex`[25] is defined as follows:

  ```
  type ReverseIndex blk = Map (HeaderHash blk) (InternalBlockInfo blk)

  data InternalBlockInfo blk = InternalBlockInfo {
        ibiFile        :: !FsPath
      , ibiBlockOffset :: !BlockOffset
      , ibiBlockSize   :: !BlockSize
      , ibiBlockInfo   :: !(BlockInfo blk)
      , ibiNestedCtxt  :: !(SomeSecond (NestedCtxt Header) blk)
      }
  ```

  In addition to the `BlockInfo` that `getBlockInfo` should return, we also store in which file the block is stored, the offset in the file, the size of the block, and the nested context (see section 6.1.1).

  The second index, called the `SuccessorsIndex` is defined as follows:

  ```
  type SuccessorsIndex blk = Map (ChainHash blk) (Set (HeaderHash blk))
  ```

  Both indices are updated when new blocks are added and when blocks are removed due to garbage collection, see section 9.2.1.

  The `Map` type used is a strict ordered map from the standard `containers` package. As for any data that is stored as long-lived state, we use strict data types to avoid space leaks. We opt for an ordered map, i.e., a sized balanced binary tree, instead of a hashing-based map to avoid hash collisions. If an attacker manages to feed us blocks that are hashed to the same bucket in the hash map, the performance will deteriorate. An ordered map is not vulnerable to this type of attack.

- Besides the mappings we discussed above, the in-memory state of the Volatile DB consists of the path, file handle, and offset into the file to which new blocks will be appended. We store this state, a pure data type, in a *read-append-write lock*, which we discuss in section 9.2.2.

- To read a block, header, or any other block component from the Volatile DB, we obtain read access to the state (see section 9.2.2) and look up the `InternalBlockInfo` corresponding to the hash in the `ReverseIndex`. The found `InternalBlockInfo` contains the file path, the block offset, and the block size, which is all what is needed to read the block. To read the header, we can use the file

[25] In a sense, this is the reverse of the mapping from file to `FileInfo`, hence the name `ReverseIndex`.

path, the block offset, the nested context (see section 6.1.1), the header offset, and header size. The other block components can also be derived from the `InternalBlockInfo`.

- Note that unlike the Immutable DB, the Volatile DB does not maintain CRC32 checksums of the stored blocks to detect corruption. Instead, after reading a block from the Volatile DB and before copying it to the Immutable DB, we validate the block using the `nodeCheckIntegrity` method, as described in section 8.2.3.

9.2.1 Garbage collection

TODO

Sync with section 12.5.

As mentioned above, when a garbage collection for slot s is triggered, all blocks with a slot less than s should be removed from the Volatile DB.

For simplicity and following our robust append-only approach, we do not modify files in-place during garbage collection. Either all the blocks in a file have a slot number less than s and it can be deleted atomically, or at least one block has a slot number greater or equal to s and we do *not* delete the file. Checking whether a file can be garbage collected is simple and happens in constant time: the `maxSlotNo` field of `FileInfo` is compared against s.

The default for blocks per file is currently 1000. Let us now calculate what the effect of this number is on garbage collection. We will call blocks that with a slot older than s *garbage*. Garbage blocks that can be deleted because they are in a file only containing garbage are *collected garbage*. Garbage blocks that cannot yet be deleted because there is a non-garbage block in the same file are *uncollected garbage*.

The lower the number of blocks per file, the fewer uncollected garbage there will be, and vice versa. In the extreme case, a single block is stored per file, resulting in no uncollected garbage, i.e., a garbage collection rate of 100%. The downside is that for each new block to add, a new file will have to be created, which is less efficient than appending to an already open file. It will also result in lots of tiny files.

The other extreme is to have no bound on the number of blocks per file, which will result in one single file containing all blocks. This means no garbage will ever be collected, i.e., a garbage collection rate of 0%, which is of course not acceptable.

During normal operation, roughly one block will be added every 20 seconds.[26] The security parameter k used for mainnet is 2160. This mean that if a linear chain of 2161 blocks has been added, the oldest block has become immutable and can be copied to the Immutable DB, after which it can be garbage collected. If we assume no delay between copying and garbage collection, it will take $1000 + 2160 = 3160$ blocks before the first file containing 1000 blocks will be garbage collected.

This means that in the above scenario, starting from a Volatile DB containing k blocks, after every blocksPerFile new blocks and thus corresponding garbage collections, blocksPerFile blocks will be garbage collected.

expand calculation

In practice we allow for overlap by delaying the garbage collection, which has an impact on the effective size of the Volatile DB, which we discuss in .

link ChainDB

9.2.2 Read-Append-Write lock

We use a *read-append-write* lock to store the state of the Volatile DB. This is an extension of the more common read-write lock. A RAW lock allows multiple concurrent readers, at most one appender, which is allowed to run concurrently with the readers, and at most one writer, which has exclusive access to the lock.

The `getBlockComponent` operation corresponds to *reading*, the `putBlock` operation to *appending*, and the `garbageCollect` operation to *writing*. Adding a new block can safely happen at the same time as blocks are being read. The new block will be appended to the current file or a new file will be started.

[26] When using the PBFT consensus protocol (section 4.5), exactly one block will be produced every 20 seconds. However, when using the Praos consensus protocol (section 4.6), on average there will be one block every 20 seconds, but it is natural to have a fork now and then, leading to one or more extra blocks. For the purposes of this calculation, the difference is negligible.

This does not affect any concurrent reads of other blocks in the Volatile DB. At most one block can be added at a time, as blocks are appended one-by-one to the current file. To garbage collect the Volatile DB, we must obtain an exclusive lock on the state, as we might be deleting a file while trying to read from it at the same time. During garbage collection, we ignore the current file and will thus never try to delete it. This means that, strictly speaking, it would be possible to safely append blocks and garbage collect blocks concurrently. However, for simplicity (how should the concurrent changes to the indices be resolved?), we did not pursue this.

As mentioned in section 9.2.1, it is often the case that no files can be garbage collected. As a (premature) optimisation, we first check (which is cheap) whether any files can be garbage collected before trying to obtain the corresponding, more expensive lock on the state.

9.2.3 Recovery

Whenever a file-system operation fails, or a file is missing or corrupted, we shut down the Volatile DB and consequently the whole system. When this happens, either the system's file system is no longer reliable (e.g., disk corruption), manual intervention (e.g., disk is full) is required, or there is a bug in the system. In all cases, there is no point in trying to continue operating. We shut down the system and flag the shutdown as *dirty*, triggering a full validation on the next start-up.

When opening the Volatile DB, the previous in-memory state, including the indices, is reconstructed based on the on-disk files. The block in each file are read and deserialised. There are two validation modes: a standard validation and a full validation. The difference between the two is that during a full validation, the integrity of each block is verified to detect silent corruption using the `nodeCheckIntegrity` method, as described in section 8.2.3.

When a block fails to deserialise or it is detected as a corrupt block when the full validation mode is enabled, the file is truncated to the last valid block before it. As mentioned at the start of this chapter, it is not crucial to recover every single block. Therefore, we do not try to deserialise the blocks after a corrupt one.

Chapter 10

Ledger Database

THIS PART WAS PORTED TO THE HADDOCKS

10.1 In-memory representation

The `LedgerDB`, capable of represent the last k ledger states, is an instantiation of the `AnchoredSeq` data type. This data type is implemented using the *finger tree* data structure [8] and has the following time complexities:

- Appending a new ledger state to the end in constant time.
- Rolling back to a previous ledger state in logarithmic time.
- Looking up a past ledger state by its point in logarithmic time.

One can think of a `AnchoredSeq` as a `Seq` from `Data.Sequence` with a custom *finger tree measure*, allowing for efficient lookups by point, combined with an *anchor*. When fully *saturated*, the sequence will contain k ledger states. In case of a complete rollback of all k blocks and thus ledger states, the sequence will become empty. A ledger state is still needed, i.e., one corresponding to the most recent immutable block that cannot be rolled back. The ledger state at the anchor plays this role.

When a new ledger state is appended to a fully saturated `LedgerDB`, the ledger state at the anchor is dropped and the oldest element in the sequence becomes the new anchor, as it has become immutable. This maintains the invariant that only the last k ledger states are stored, *excluding* the ledger state at the anchor. This means that in practice, $k + 1$ ledger states will be kept in memory. When fewer the `LedgerDB` contains fewer than k elements, new ones are appended without shifting the anchor until it is saturated. TODO figure?

The `LedgerDB` is parameterised over the ledger state l. Conveniently, the `LedgerDB` can implement the same abstract interface (described in section 5.1) that the ledger state itself implements. I.e., the `GetTip`, `IsLedger`, and `ApplyBlock` classes. This means that in most places, wherever a ledger state can be used, it is also possible to wrap it in a `LedgerDB`, causing it to automatically maintain a history of the last k ledger states. TODO discuss `Ap` and `applyBlock`? These are actually orthogonal to `LedgerDB` and should be separated.

Memory usage The ledger state is a big data structure that contains, amongst other things, the entire UTxO. Recent measurements[27] show that the heap size of an Allegra ledger state is around 361 MB. Fortunately, storing $k = 2160$ ledger states in memory does *not* require $2160 * 361$ MB = 779,760 MB = 761 GB. The ledger state is defined using standard Haskell data structures, e.g., `Data.Map.Strict`, which are *persistent* data structures. This means that when we update a ledger state by applying a block to it, we

[27] Using the ledger state at the block with slot number 16,976,076 and hash `af0e6cb8ead39a86`.

only need extra memory for the new and the modified data. The majority of the data will stay the same and will be *shared* with the previous ledger state.

The memory required for storing the last k ledger state is thus proportional to: the size of the oldest in-memory ledger state *and* the changes caused by the last k blocks, e.g., the number of transactions in those blocks. Compared to the 361 MB required for a single ledger state, keeping the last k ledger states in memory requires only 375 MB in total. This is only 14 MB or 3.8% more memory. Which is a very small cost.

Past design In the past, before measuring this cost, we did not keep all k past ledger states because of an ungrounded fear for the extra memory usage. The `LedgerDB` data structure had a `snapEvery` parameter, ranging from 1 to k, indicating that a snapshot, i.e., a ledger state, should be kept every `snapEvery` ledger states or blocks. In practice, a value of 100 was used for this parameter, resulting in 21–22 ledger states in memory.

The representation was much more complex, to account for these missing ledger states. More importantly, accessing a past ledger state or rewinding the `LedgerDB` to a past ledger state had a very different cost model. As the requested ledger state might not be in memory, it would have to be *reconstructed* by reapplying blocks to an older ledger state.

Consider the example below using `snapEvery` = 3. L_i indicate ledger states and $\emptyset_i$ indicate skipped ledger states. L_0 corresponds to the most recent ledger state, at the tip of the chain.

$\cdots$	L_6	$\emptyset_5$	$\emptyset_4$	L_3	$\emptyset_2$	$\emptyset_1$	L_0

When we need access to the ledger state at position 3, we are in luck and can use the available L_3. However, when we need access to the skipped ledger state at position 1, we have to do the following: we look for the most recent ledger state before $\emptyset_1$, i.e., L_3. Next, we need to reapply blocks B_2 and B_1 to it, which means we have to read those from disk, deserialise them, and apply them again.

This means that accessing a past ledger state is not a pure operation and might require disk access and extra computation. Consequently, switching to a fork might require reading and revalidating blocks that remain part of the chain, in addition to the new blocks.

As mentioned at the start of this chapter, the chain sync client also needs access to past ledger view (section 4.2.2), which it can obtain from past ledger states. A malicious peer might try to exploit it and create a chain that intersects with our chain right *before* an in-memory ledger state snapshot. In the worst case, we have to read and reapply `snapEvery` - 1 = 99 blocks. This is not acceptable as the costs are asymmetric and in the advantage of the attacker, i.e., creating and serving such a header is much cheaper than reconstructing the required snapshot. At the time, we solved this by requiring ledger states to store snapshots of past ledger views. The right past ledger view could then be obtained from the current ledger state, which was always available in memory. However, storing snapshots of ledger views within a single ledger state is more complex, as we are in fact storing snapshots *within* snapshots. The switch to keep all k past ledger states significantly simplified the code and sped up the look-ups.

Future design It is important to note that in the future, this design will have to change again. The UTxO and, consequently, the ledger state are expected to grow in size organically. This growth will be accelerated by new features added to the ledger, e.g., smart contracts. At some point, the ledger state will be so large that keeping it in its entirety in memory will no longer be feasible. Moreover, the cost of generating enough transactions to grow the current UTxO beyond the expected memory limit might be within reach for some attackers. Such an attack might cause a part of the network to be shut down because the nodes in question are no longer able to load the ledger state in memory without running against the memory limit.

For these reasons, we plan to revise our design in the future, and start storing parts of the ledger state on disk again.

10.2 On-disk

The `LedgerDB.OnDisk` module provides functions to write a ledger state to disk and read a ledger state from disk. The `EncodeDisk` and `DecodeDisk` classes from section 6.1 are used to (de)serialise the ledger state to or from CBOR. Because of its large size, we read and deserialise the snapshot incrementally. TODO

which ledger state to take a snapshot from is determined by the Chain DB. I.e., the background thread that copies blocks from the Volatile DB to the Immutable DB will call the `onDiskShouldTakeSnapshot` function, and if it returns **`True`**, a snapshot will be taken. double-check whether we're actually taking a snapshot of the right ledger state. TODO

10.2.1 Disk policy

The disk policy determines how many snapshots should be stored on disk and when a new snapshot of the ledger state should be written to disk. TODO

worth discussing? We would just be duplicating the existing documentation.

10.2.2 Initialisation

During initialisation, the goal is to construct an initial `LedgerDB` that is empty except for the ledger state at the anchor, which has to correspond to the immutable tip, i.e., the block at the tip of the Immutable DB (chapter 8).

Ideally, we can construct the initial `LedgerDB` from a snapshot of the ledger state that we wrote to disk. Remember that updating a ledger state with a block is not invertible: we can apply a block to a ledger state, but we cannot "unapply" a block to a ledger state. This means the snapshot has to be at least as old as the anchor. A snapshot matching the anchor can be used as is. A snapshot older than the anchor can be used after reapplying the necessary blocks. A snapshot newer than the anchor can *not* be used, as we cannot unapply blocks to get the ledger state corresponding to the anchor. This is the reason why we only take snapshots of an immutable ledger state, i.e., of the anchor of the `LedgerDB` (or older).

Constructing the initial `LedgerDB` proceeds as follows:

1. If any on-disk snapshots are available, we try them from new to old. The newer the snapshot, the fewer blocks will need to be reapplied.

2. We deserialise the snapshot. If this fails, we try the next one.

3. If the snapshot is of the ledger state corresponding to the immutable tip, we can use the snapshot for the anchor of the `LedgerDB` and are done.

4. If the snapshot is newer than the immutable tip, we cannot use it and try the next one. This situation can arise not because we took a snapshot of a ledger state newer than the immutable tip, but because the Immutable DB was truncated.

5. If the snapshot is older than the immutable tip, we will have to reapply the blocks after the snapshot to obtain the ledger state at the immutable tip. If there is no (more) snapshot to try, we will have to reapply *all blocks* starting from the beginning of the chain to obtain the ledger state at the immutable tip, i.e., the entire immutable chain. The blocks to reapply are streamed from the Immutable DB, using an iterator (section 8.1.2).

   Note that we can *reapply* these blocks, which is quicker than applying them (see section 10.3), as the existence of a snapshot newer than these blocks proves[28] that they have been successfully applied in the past.

[28] Unless the on-disk database has been tampered with, but this is not an attack we intend to protect against, as this would mean the machine has already been compromised.

Reading and applying blocks is costly. Typically, very few blocks need to be reapplied in practice. However, there is one exception: when the serialisation format of the ledger state changes, all snapshots (written using the old serialisation format) will fail to deserialise, and all blocks starting from genesis will have to be reapplied. To mitigate this, the ledger state decoder is typically written in a backwards-compatible way, i.e., it accepts both the old and new serialisation format.

10.3 Maintained by the Chain DB

TODO

move to Chain DB chapter?

The `LedgerDB` is a pure data structure. The Chain DB (see chapter 12) maintains the current `LedgerDB` in a `StrictTVar`. The most recent element in the `LedgerDB` is the current ledger state. Because it is stored in a `StrictTVar`, the current ledger state can be read and updated in the same `STM` transaction as the current chain, which is also stored in a `StrictTVar`.

The `ChainDB.Impl.LgrDB`[29] is responsible for maintaining the current ledger state. Besides this responsibility, it also integrates the Ledger DB with other parts of the Chain DB.

Moreover, it remembers which blocks have been successfully applied in the past. When such a block needs to be validated again, e.g., because we switch again to the same fork containing the block, we can *reapply* the block instead of *applying* it (see section 5.1.2). Because the block has been successfully applied in the past, we know the block is valid, which means we can skip some of the more expensive checks, e.g., checking the hashes, speeding up the process of validating the block. Note that a block can only be applied to a single ledger state, i.e., the ledger state corresponding to the predecessor of the block. Consequently, it suffices to remember whether a block was valid or not, there is no need to remember with respect to which ledger state it was valid.

To remember which blocks have been successfully applied in the past, we store the points of the respective blocks in a set. Before validating a block, we look up its point in the set, when present, we can reapply the block instead of applying it. To stop this set from growing without bound, we garbage collect it the same way the Volatile DB is garbage collected, see section 12.5. When a block has a slot older than the slot number of the most recent immutable block, either the block is already immutable or it is part of a fork that we will never consider again, as it forks off before the immutable block. The block in question will never have to be validated again, and so it is not necessary to remember whether we have already applied it or not.

slot number vs block number

[29] In the past, we had similar modules for the `VolatileDB` and `ImmutableDB`, i.e., `VolDB` and `ImmDB`. The former were agnostic of the `blk` type and the latter instantiated the former with the `blk` type. However, in hindsight, unifying the two proved to be simpler and was thus done. The reason why a separate `LgrDB` still exists is mainly because it needs to wrap the pure `LedgerDB` in a `StrictTVar`.

Chapter 11

Chain Selection

Chain selection is one of the central responsibilities of the chain database (chapter 12). It of course depends on chain selection as it is defined by the consensus protocol (section 4.2.1), but needs to take care of a lot of operational concerns. In this chapter we will take a closer look at the implementation of chain selection in the chain database, and state some properties and sketch some proofs to motivate it.

11.1 Comparing anchored fragments

11.1.1 Introduction

Recall from section 4.1.1 that while in the literature chain selection is defined in terms of comparisons between entire chains, we instead opted to model it in terms of a comparison between the *headers* at the tip of those chains (or rather, a *view* on those headers defined by the specific consensus protocol).

We saw in section 7.2 (specifically, section 7.2.1) that the consensus layer stores chain fragments in memory (the most recent headers on a chain), both for the node's own current chain as well as for upstream nodes (which we refer to as "candidate chains"). Defining chain selection in terms of fragments is straight-forward when those fragments are non-empty: we simply take the most recent header, extract the view required by the consensus protocol (section 4.3), and then use the consensus protocol's chain selection interface to compare them. The question is, however, how to compare two fragments when one (or both) of them is *empty*. This problem is more subtle than it might seem at first sight, and requires careful consideration.

We mentioned in section 4.1.1 that consensus imposes a fundamental assumption that the strict extension of a chain is always (strictly) preferred over that chain (assumption 4.1), and that consequently we *always* prefer a non-empty chain over an empty one (and conversely we *never* prefer an empty chain over a non-empty one). However, chain fragments are mere proxies for their chains, and the fragment might be empty even if the chain is not. This means that in principle it's possible we do not prefer a non-empty fragment over an empty one, or indeed prefer an empty fragment over a non-empty one. However, when a fragment is empty, we cannot rely on the consensus protocol's chain selection because we have no header to give it.

Let's consider under which conditions these fragments might be empty:

Our fragment Our own fragment is a path through the volatile database, anchored at the tip of the immutable database (section 7.2.1). Under normal circumstances, it will be empty only if our *chain* is empty; we will refer to such empty fragments as *genuinely empty*.[30] However, our fragment can also be empty even when our chain is not, if due to data loss the volatile database is empty (or contains no blocks that fit onto the tip of the immutable database).

[30] We can distinguish between an empty fragment of a non-empty chain and a (necessarily) empty fragment of an empty chain by looking at the anchor point: if it is the genesis point, the chain must be empty.

Candidate fragment A *genuinely* empty candidate fragment, representing an empty candidate chain, is never preferred over our chain. Unfortunately, however, the candidate fragment as maintained by the chain sync client (chapter 14) can essentially be empty at any point due to the way that a switch-to-fork is implemented in terms of rollback followed by roll forward: after a maximum rollback (and before the roll forward), the candidate fragment is empty.

11.1.2 Precondition

Since neither of these circumstances can be avoided, we must therefore impose a precondition for chain selection between chain fragments to be definable:

Definition 11.1 (Precondition for comparing chain fragments)**.** The two fragments must either both be non-empty, or they must intersect.

In this chapter, we establish this precondition in two different ways:

1. When we construct candidates chains (potential chains that we may wish to replace our own chain with), those candidate chains must intersect with our own chain within k blocks from its tip; after all, if that is not the case, we would induce a roll back of more than k blocks (section 4.1.2).
2. When we compare fragments to each other, we only compare fragments from a set of fragments that are all anchored at the same point (i.e., the anchor of all fragments in the set is the same, though it might be different from the anchor of our current fragment). Since they are all anchored at the same point, they trivially all intersect with each other.

There is one more use of fragment selection, which is rather more subtle; we will come back to this in section 14.4.

TODO

TODO: Throughout we are talking about *anchored* fragments here. We should make sure that we discuss those somewhere.

11.1.3 Definition

We will now show that this precondition suffices to compare two fragments, whether or not they are empty; we'll consider each case in turn.

Both fragments empty Since the two fragments must intersect, that intersection point can only be the two anchor points, which must therefore be equal. This means that two fragments represent the same chain: neither fragment is preferred over the other.

First fragment non-empty, second fragment empty Since the two fragments must intersect, that intersection can only be the anchor of the second fragment, which can lie anywhere on the first fragment.

- If it lies at the *tip* of the first fragment, the two fragments represent the same chain, and neither is preferred over the other.
- If it lies *before* the tip of first fragment, the first fragment is a strict extension of the second, and is therefore is preferred over the second.

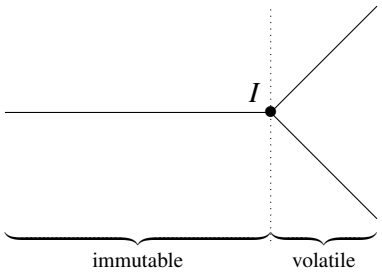
First fragment empty, second fragment non-empty This case is entirely symmetric to the previous; if the intersection is the tip of the second fragment, the fragments represent the same chain. Otherwise, the second fragment is a strict extension of the first, and is therefore preferred.

Both fragments non-empty In this case, we can simply use the consensus protocol chain selection API to compare the two most recent headers on both fragments.

Note that this relies critically on the "prefer extension" rule (assumption 4.1).

11.2 Preliminaries

Recall from section 7.1 that the immutable database stores a linear chain, terminating in the *tip I* of the immutable database. The volatile database stores a (possibly fragmented) tree of extensions to that chain:



The node's *current chain* is stored in memory as a chain fragment through the volatile database, anchored at I. When we start up the node, the chain database must find the best possible path through the volatile database and adopt that as our current fragment; every time a new block is added to the volatile database, we have to recompute the new best possible path. In other words, we maintain the following invariant:

Definition 11.2 (Current chain invariant)**.** The current chain is the best possible path through the volatile DB.

"Best" of course is according to the chain selection rule defined by the consensus protocol (section 4.2.1). In this section we describe how the chain database establishes and preserves this invariant.

11.2.1 Notation

So far we have been relatively informal in our description of chain selection, but in order to precisely describe the algorithm and state some of its properties, we have to introduce some notation.

Definition 11.3 (Chain selection)**.** We will model chain selection as a transitive binary relation ($\sqsubset$) between valid chains (it is undefined for invalid chains), and let $C \sqsubseteq C'$ if and only if $C \sqsubset C'$ or $C = C'$. It follows that ($\sqsubseteq$) is a partial order (reflexive, antisymmetric, and transitive).

For example, the simple "prefer longest chain" chain selection rule could be given as

$$C \sqsubset C' \qquad \text{iff} \qquad \text{length } C < \text{length } C' \tag{longest chain rule}$$

In general of course the exact rule depends on the choice of consensus protocol. Assumption 4.1 (section 4.1.1) can now be rephrased as

$$\forall C, B.\ C \sqsubset (C \triangleright B) \tag{11.1}$$

We will not be comparing whole chains, but rather chain fragments (we will leave the anchor of fragments implicit):

Definition 11.4 (Fragment selection)**.** We lift $\sqsubset$ to chain fragments in the manner described in section 11.1; this means that $\sqsubset$ is undefined for two fragments if they do not intersect (section 11.1.2).

We also lift $\sqsubseteq$ to *sets* of fragments, intuitively indicating that a particular fragment is the "best choice" out of a set $\mathcal{S}$ of candidate fragments:

Definition 11.5 (Optimal candidate)**.**

$$\mathcal{S} \sqsubseteq F \qquad \text{iff} \qquad \nexists F' \in \mathcal{S}.\ F \sqsubset F'$$

(in other words, if additionally $F \in \mathcal{S}$, then F is a maximal element of C). This inherits all the preconditions of $\sqsubseteq$ on chains and fragments.

Finally, we will introduce some notation for *computing* candidate fragments:[31]

Definition 11.6 (Construct set of candidates)**.** Given some set of blocks V, and some anchor A (with A either a block or the genesis point),

$$\mathsf{candidates}_A(V)$$

is the set of chain fragments anchored at A using blocks picked from V.

By construction all fragments in $\mathsf{candidates}_A(V)$ have the same anchor, and hence all intersect (at A); this will be important for the use of the $\sqsubseteq$ operator.

11.2.2 Properties

In the following we will use $(F \triangleright B)$ to denote appending block B to chain F, and lift this notation to sets, so that for some set of blocks $\mathcal{B}$ we have

$$F \triangleright \mathcal{B} = \{F \triangleright B \mid B \in \mathcal{B}\}$$

Lemma 11.1 (Properties of the set of candidates)**.** The set of candidates computed by $\mathsf{candidates}_A(V)$ has the following properties.

1. It is prefix closed:

$$\forall F, B.\ \text{if} \quad (F \triangleright B) \in \mathsf{candidates}_A(V) \quad \text{then} \quad F \in \mathsf{candidates}_A(V)$$

2. If we add a new block into the set, we can append that block to existing candidates (where it fits):

$$\text{if} \quad F \in \mathsf{candidates}_A(V) \quad \text{then} \quad F \triangleright B \in \mathsf{candidates}_A(V \cup \{B\})$$

   provided F can be extended with B.

3. Adding blocks doesn't remove any candidates:

$$\mathsf{candidates}_A(V) \subseteq \mathsf{candidates}_A(V \cup \{B\})$$

4. If we add a new block, then any new candidates must involve that new block:

$$\text{if} \quad F \in \mathsf{candidates}_A(V \cup \{B\}) \quad \text{then} \quad F \in \mathsf{candidates}_A(V) \text{ or } F = (\ldots \triangleright B \triangleright \ldots)$$

The next lemma says that if we have previously found some optimal candidate F, and subsequently learn of a new block B (where B is a direct or indirect extension of F), it suffices to find a locally optimal candidate *amongst the candidates that involve B*; this new candidate will also be a globally optimal candidate.

Lemma 11.2 (Focus on new block)**.** Suppose we have F, F_{new} such that

1. $\mathsf{candidates}_A(V) \sqsubseteq F$
2. $(\mathsf{candidates}_A(V \cup \{B\}) \setminus \mathsf{candidates}_A(V)) \sqsubseteq F_{new}$
3. $F \sqsubseteq F_{new}$

Then

$$\mathsf{candidates}_A(V \cup \{B\}) \sqsubseteq F_{new}$$

Proof. Suppose there exists $F' \in \mathsf{candidates}_A(V \cup \{B\})$ such that $F_{new} \sqsubset F'$. By transitivity and assumption 3, $F \sqsubset F'$. As shown in lemma 11.1 (item 4), there are two possibilities:

- $F' \in \mathsf{candidates}_A(V)$, which would violate assumption item 1, or
- F' must contain block B, which would violate assumption item 2.

□

[31] In order to compute these candidates efficiently, the volatile database must support a "forward chain index", able to efficiently answer the question "which blocks succeed this one?".

11.3 Initialisation

The initialisation of the chain database proceeds as follows.

1. Initialise the immutable database, determine its tip I, and ask the ledger DB for the corresponding ledger state L (see section 10.2.2).

2. Compute the set of candidates anchored at the immutable database's tip I using blocks from the volatile database V

   $$\mathsf{candidates}_I(V)$$

   ignoring known-to-be-invalid blocks (if any; see section 12.3) and order them using ($\sqsubset$) so that we process better candidates first.[32] Candidates that are strict prefixes of other candidates can be ignored (as justified by the "prefer extension" assumption, assumption 4.1);[33] we may reconsider some of these prefixes if we subsequently discover invalid blocks (see item 3).

3. Not all of these candidates may be valid, because the volatile database stores blocks whose *headers* have been validated, but whose *bodies* are still unverified (other than to check that they correspond to their headers). We therefore validate each candidate chain fragment, starting with L (the ledger state at the tip of the immutable database) each time.[34]

   As soon as we find a candidate that is valid, we adopt it as our current chain. If we find a candidate that is *invalid*, we mark the invalid block[35] (unless it is invalid due to potential clock skew, see section 11.5), and go back to step 2. It is important to recompute the set of candidates after marking some blocks as invalid because those blocks may also exist in other candidates and we do not know how the valid prefixes of those candidates should now be ordered.

11.4 Adding new blocks

When a new block B is added to the chain database, we need to add it to the volatile DB and recompute our current chain. We distinguish between the following different cases.

Before we process the new block, we first run chain selection on any blocks that had previously been temporarily shelved because their slot number was (just) ahead of the wallclock (section 11.5). We do this independent of what we do with the new block.[36]

The implementation `addBlock` additionally provides client code with various notifications throughout the process ("block added", "chain selection run", etc.). We will not describe these notifications here.

11.4.1 Ignore

We can just ignore the block if any of the following is true.

- The block is already in the immutable DB, *or* it belongs to a branch which forks more than k blocks away from our tip, i.e.[37]

  $$\mathtt{blockNo}(B) \leq \mathtt{blockNo}(I)$$

[32] Technically speaking we should *first* validate all candidates, and only then apply selection to the valid chains. We perform chain selection first, because that is much cheaper. Both approaches are semantically equivalent, since `sortBy f . filter p = filter p . sortBy f` due to the stability of `sortBy`.

[33] The implementation does not compute candidates, but rather "maximal" candidates, which do not include such prefixes.

[34] We make no attempt to share ledger states between candidates, even if they share a common prefix, trading runtime performance for lower memory pressure.

[35] There is no need to mark any successors of invalid blocks; see lemma 12.1.

[36] In a way, calls to `addBlock` are how the chain database sees time advance. It does not rely on slot length to do so, because slot length is ledger state dependent.

[37] The check is a little more complicated in the presence of EBBs (chapter 23). This is relevant if we switch to an alternative fork after a maximum rollback, and that alternative fork starts with an EBB. It is also relevant when due to data corruption the volatile database is empty and the first block we add when we continue to sync the chain happens to be an EBB.

We could distinguish between between the block being on our chain or on a distant fork by doing a single query on the immutable database, but it does not matter: either way we do not care about this block.

We don't expect the chain sync client to feed us such blocks under normal circumstances, though it's not impossible: by the time a block is downloaded it's conceivable, albeit unlikely, that that block is now older than k.

- The block was already in the volatile database, i.e.

$$B \in V$$

- The block is known to be invalid (section 12.3).

11.4.2 Add to current chain

Let B_{pred} be the predecessor block of B. If B fits onto the end of our current fragment F (and hence onto our current chain) F, i.e.

- F is empty, and $B_{pred} = I$ (where I must necessarily also be the anchor of the fragment), or
- $\exists F'.\ F = F' \triangleright B_{pred}$

then any new candidates must be equal to or an extension of $F \triangleright B$ (lemma 11.1, item 4); this set is computed by

$$(F \triangleright B \triangleright \mathsf{candidates}_B(V \cup \{B\}))$$

Since all candidates would be strictly preferred over F (since they are extensions of F), by lemma 11.2 it suffices to pick the best candidate amongst these extensions. Apart from the starting point, chain selection then proceeds in the same way as when we are initialising the database (section 11.3).

This case takes care of the common case where we just add a block to our chain, as well as the case where we stay with the same branch but receive some blocks out of order. Moreover, we can use the *current* ledger state as the starting point for validation.

11.4.3 Store, but don't change current chain

When we are missing one of the (transitive) predecessors of the block, we store the block but do nothing else. We can check this by following back pointers until we reach a block B' such that $B' \notin V$ and $B' \neq I$. The cost of this is bounded by the length of the longest fragment in the volatile DB, and will typically be low; moreover, the chain fragment we are constructing this way will be used in the switch-to-fork case (section 11.4.4).[38]

At this point we *could* do a single query on the immutable DB to check if B' is in the immutable DB or not. If it is, then this block is on a distant branch that we will never switch to, and so we can ignore it. If it is not, we may or may not need this block later and we must store it; if it turns out we will never need it, it will eventually be garbage collected (section 12.5).

An alternative and easier approach is to omit the check on the immutable DB, simply assuming we might need the block, and rely on garbage collection to eventually remove it if we don't. This is the approach we currently use.

[38]The function that constructs these fragments is called `isReachable`.

11.4.4 Switch to a fork

If none of the cases above apply, we have a block B such that

1. $B \notin V$
2. $\texttt{blockNo}(B) > \texttt{blockNo}(I)$ (and hence B cannot be in the immutable DB)
3. For all transitive predecessors B' of B we have $B' \in V$ or $B' = I$. In other words, we must have a fragment
$$F_{prefix} = I \triangleright \ldots \triangleright B$$
in $\mathsf{candidates}_I(V \cup \{B\})$.
4. (Either F is empty and $B_{pred} \neq I$, or) $\exists F', B'.\ F = F' \triangleright B'$ where $B' \neq B_{pred}$; i.e., block does not fit onto current chain.[39]

(This list is just the negation of the conditions we handled in the sections above.) We proceed in similar fashion to the case when the block fit onto the tip of our chain (section 11.4.2). The new candidates in $\mathsf{candidates}_I(V \cup \{B\})$ must involve B (lemma 11.1, item 4), which in this case means they must all be extensions of F_{prefix}; we can compute these candidates using[40]

$$I \triangleright \ldots \triangleright B \triangleright \mathsf{candidates}_B(V \cup \{B\})$$

Not all of these fragments might be preferred over the current chain; we filter those out.[41] We then proceed as usual, considering each of the remaining fragments in ($\sqsubseteq$) order, and appeal to lemma 11.2 again to conclude that the fragment we find in this way will be an optimal candidate across the entire volatile database.

11.5 In-future check

As we saw in section 11.2, the chain DB performs full block validation during chain selection. When we have validated a block, we then do one additional check, and verify that the block's slot number is not ahead of the wallclock time (for a detailed discussion of why we require the block's ledger state for this, see chapter 17, especially section 17.6.2). If the block is far ahead of the wallclock, we treat this as any other validation error and mark the block as invalid.

Marking a block as invalid will cause the network layer to disconnect from the peer that provided the block to us, since non-malicious (and non-faulty) peers should never send invalid blocks to us. It is however possible that an upstream peer's clock is not perfectly aligned with us, and so they might produce a block which *we* think is ahead of the wallclock but *they* do not. To avoid regarding such peers as malicious, the chain database supports a configurable *permissible clock skew*: blocks that are ahead of the wallclock by an amount less than this permissible clock skew are not marked as invalid, but neither will chain selection adopt them; instead, they simply remain in the volatile database available for the next chain selection.

It is constructive to consider what happens if *our* clock is off, in particular, when it is slow. In this scenario *every* (or almost every) block that the node receives will be considered to be in the future. Suppose we receive two consecutive blocks A and B. When we receive A, chain selection runs, we find that A is ahead of the clock but within the permissible clock skew, and we don't adopt it. When we then

[39] Item 3 rules out the first option: if $B_{pred} \neq I$ then we must have $B_{pred} \in V$ and moreover this must form some kind of chain back to I; this means that the preferred candidate cannot be empty.

[40] The implementation of the chain database actually does not construct fragments that go back to I, but rather to the intersection point with the current chain. This can be considered to be an optimisation of what we describe here.

[41] Recall that the current chain gets special treatment: when two candidates are equally preferable, we can pick either one, but when a candidate and the current chain are equally preferable, we must stick with the current chain.

receive B, chain selection runs again, we now discover the A, B extension to our current chain; during validation we cut off this chain at B because it is ahead of the clock, but we adopt A because it is now valid. In other words, we are always behind one block, adopting each block only when we receive the *next* block.

11.6 Sorting

In this chapter we have modelled chain selection as a partial order ($\sqsubseteq$). This suffices for the formal treatment, and in theory also suffices for the implementation. However, at various points during the chain selection process we need to *sort* candidates in order of preference. We can of course sort values based on a preorder only (topological sorting), but we can do slightly better. Recall from section 4.2.1 that we require that the `SelectView` on headers must be a total order. We can therefore define

Definition 11.7 (Same select view)**.** Let $C \precsim C'$ if the select view at the tip of C is less than or equal to the select view at the tip of C'.

($\precsim$) forms a total preorder (though not a partial order); if $C \precsim C'$ *and* $C' \precsim C$ then the select views at the tips of C and C' are equal (though they might be different chains, of course). Since $C \precsim C'$ implies $C' \not\sqsubset C$, we can use this preorder to sort candidates (in order words, we will sort them *on* their select view, in Haskell-parlance).

Chapter 12

Chain Database

TODO: This is currently a disjoint collection of snippets. TODO

12.1 Union of the Volatile DB and the Immutable DB

As discussed in section 7.1, the blocks in the Chain DB are divided between the Volatile DB (chapter 9) and the Immutable DB (chapter 8). Yet, it presents a unified view of the two databases. Whereas the Immutable DB only contains the immutable chain and the Volatile DB the volatile *parts* of multiple forks, by combining the two, the Chain DB contains multiple forks.

12.1.1 Looking up blocks

Just like the two underlying databases the Chain DB allows looking up a `BlockComponent` of a block by its point. By comparing the slot number of the point to the slot of the immutable tip, we could decide in which database to look up the block. However, this would not be correct: the point might have a slot older than the immutable tip, but refer to a block not in the Immutable DB, i.e., a block on an older fork. More importantly, there is a potential race condition: between the time at which the immutable tip was retrieved and the time the block is retrieved from the Volatile DB, the block might have been copied to the Immutable DB and garbage collected from the Volatile DB, resulting in a false negative. Nevertheless, the overlap between the two makes this scenario very unlikely.

For these reasons, we look up a block in the Chain DB as follows. We first look up the given point in the Volatile DB. If the block is not in the Volatile DB, we fall back to the Immutable DB. This means that if, at the same, a block is copied from the Volatile DB to the Immutable DB and garbage collected from the Volatile DB, we will still find it in the Immutable DB. Note that failed lookups in the Volatile DB are cheap, as no disk access is required.

12.1.2 Iterators

Similar to the Immutable DB (section 8.1.2), the Chain DB allows streaming blocks using iterators. We only support streaming blocks from the current chain or from a recent fork. We *do not* support streaming from a fork that starts before the current immutable tip, as these blocks are likely to be garbage collected soon. Moreover, it is of no use to us to serve another node blocks from a fork we discarded.

We might have to stream blocks from the Immutable DB, the Volatile DB, or from both. If the end bound is older or equal to the immutable tip, we simply try to open an Immutable DB iterator with the given bounds. If the end bound is newer than the immutable tip, we construct a path of points (see `filterByPredecessor` in section 9.1) connecting the end bound to the start bound. This path is either entirely in the Volatile DB or it is partial because a block is missing from the Volatile DB. If the missing block is the tip of the Immutable DB, we will have to stream from the Immutable DB in addition to the Volatile DB. If the missing block is not the tip of the Immutable DB, we consider the range to be invalid.

In other words, we allow streaming from both databases, but only if the immutable tip is the transition point between the two, it cannot be a block before the tip, as that would mean the fork is too old. TODO

Image?

To stream blocks from the Volatile DB, we maintain the constructed path of points as a list in memory and look up the corresponding block (component) in the Volatile DB one by one.

Consider the following scenario: we open a Chain DB iterator to stream the beginning of the current volatile chain, i.e., the blocks in the Volatile DB right after the immutable tip. However, before streaming the iterator's first block, we switch to a long fork that forks off all the way back at our immutable tip. If that fork is longer than the previous chain, blocks from the start of our chain will be copied from the Volatile DB to the Immutable DB, advancing the immutable tip. This means the blocks the iterator will link stream are now part of a fork older than k. In this new situation, we would not allow opening an iterator with the same range as the already-opened iterator. However, we do allow streaming these blocks using the already opened iterator, as the blocks to stream are unlikely to have already been garbage collected. Nevertheless, it is still theoretically possible[42] that such a block has already been garbage collected. For this reason, the Chain DB extends the Immutable DB's `IteratorResult` type (see section 8.1.2) with the `IteratorBlockGCed` constructor:

```
data IteratorResult blk b =
    IteratorExhausted
  | IteratorResult b
  | IteratorBlockGCed (RealPoint blk)
```

There is another scenario to consider: we stream the blocks from the start of the current volatile chain, just like in the previous scenario. However, in this case, we do not switch to a fork, but our chain is extended with new blocks, which means blocks from the start of our volatile chain are copied from the Volatile DB to the Immutable DB. If these blocks have been copied and garbage collected before the iterator is used to stream them from the Volatile DB (which is unlikely, as explained in the previous scenario), the iterator will incorrectly yield `IteratorBlockGCed`. Instead, when a block that was planned to be streamed from the Volatile DB is missing, we first look in the Immutable DB for the block in case it has been copied there. After the block copied to the Immutable has been streamed, we continue with the remaining blocks to stream from the Volatile DB. It might be the case that the next block has also been copied and garbage collected, requiring another switch to the Immutable DB. In the theoretical worst case, we have to switch between the two databases for each block, but this is nearly impossible to happen in practice.

12.1.3 Followers

In addition to iterators, the Chain DB also supports *followers*. Unlike an iterator, which is used to request a static segment of the current chain or a recent fork, a follower is used to follow the *current chain*. Either from the start of from a suggested more recent point. Unlike iterators, followers are dynamic, they will follow the chain when it grows or forks. A follower is pull-based, just like its primary user, the chain sync server (see section 15.2). This avoids the need to have a growing queue of changes to the chain on the server side in case the client side is slower.

The API of a follower is as follows:

```
data Follower m blk a = Follower {
      followerInstruction         :: m (Maybe (ChainUpdate blk a))
    , followerInstructionBlocking :: m (ChainUpdate blk a)
    , followerForward             :: [Point blk] -> m (Maybe (Point blk))
    , followerClose               :: m ()
    }
```

[42] This is unlikely, as there is a delay between copying and garbage collection (see section 12.5.1) and there are network time-outs on the block fetch protocol, of which the server-side (see section 15.4) is the primary user of Chain DB iterators.

The `a` parameter is the same `a` as the one in `BlockComponent` (see section 8.1.1), as a follower for any block component `a` can be opened.

A follower always has an implicit position associated with it. The `followerInstruction` operation and its blocking variant allow requesting the next instruction w.r.t. the follower's implicit position, i.e., a `ChainUpdate`:

```
data ChainUpdate block a =
    AddBlock a
  | RollBack (Point block)
```

The `AddBlock` constructor indicates that to follow the current chain, the follower should extend its chain with the given block (component). Switching to a fork is represented by first rolling back to a certain point (`RollBack`), followed by at least as many new blocks (`AddBlock`) as blocks that have been rolled back. If we were to represent switching to a fork using a constructor like:

```
  | SwitchToFork (Point block) [a]
```

we would need to have many blocks or block components in memory at the same time.

These operations are implemented as follows. In case the follower is looking at the immutable part of the chain, an Immutable DB iterator is used and no rollbacks will be encountered. When the follower has advanced into the volatile part of the chain, the in-memory fragment containing the last k headers is used (see section 7.2). Depending on the block component, the corresponding block might have to be read from the Volatile DB.

When a new chain has been adopted during chain selection (see section 11.4), all open followers that are looking at the part of the current chain that was rolled back are updated so that their next instruction will be the correct `RollBack`. By definition, followers looking at the immutable part of the chain will be unaffected.

By default, a follower will start from the very start of the chain, i.e., at genesis. Accordingly, the first instruction will be an `AddBlock` with the very first block of the chain. As mentioned, the primary user of a follower is the chain sync server, of which the clients in most cases already have large parts of the chain. The `followerForward` operation can be used in these cases to find a more recent intersection from which the follower can start. The client will sent a few recent points from its chain and the follower will try to find the most recent of them that is on our current chain. This is implemented by looking up blocks by their point in the current chain fragment and the Immutable DB.

Followers are affected by garbage collection similarly to how iterators are (section 12.1.2): when the implicit position of the follower is in the immutable part of the chain, an Immutable DB iterator with a static range is used. Such an iterator is not aware of blocks appended to the Immutable DB since the iterator was opened. This means that when the iterator reaches its end, we first have to check whether more blocks have been appended to the Immutable DB. If so, a new iterator is opened to stream these blocks. If not, we switch over to the in-memory fragment.

12.2 Block processing queue

Discuss the chain DB's block processing queue, the future/promises/events, concurrency concerns, etc.

Discuss the problem of the effective queue size (#2721).

12.3 Marking invalid blocks

The chain database keeps a set of hashes of known-to-be-invalid blocks. This information is used by the chain sync client (chapter 14) to terminate connections to nodes with a chain that contains an invalid block.

Lemma 12.1. When the chain database discovers an invalid block X, it is sufficient to mark only X; there is no need to additionally mark any successors of X.

Proof (sketch). The chain sync client maintains a chain fragment corresponding to some suffix of the upstream node's chain, and it preserves an invariant that that suffix must intersect with the node's own current chain. It can therefore never be the case that the fragment contains a successor of X but not X itself: since X is invalid, the node will never adopt it, and so a fragment that intersects the node's current chain and includes a successor of X *must* also contain X. □

TODO: We should discuss how this relates to GC (section 12.5). TODO

12.4 Effective maximum rollback

The maximum rollback we can support is bound by the length of the current fragment. This will be less than k only if

- We are near genesis and the immutable database is empty, or
- Due to data corruption the volatile database lost some blocks

Only the latter case is some cause for concern: we are in a state where conceptually we *could* roll back up to k blocks, but due to how we chose to organise the data on disk (the immutable/volatile split) we cannot. One option here would be to move blocks *back* from the immutable DB to the volatile DB under these circumstances, and indeed, if there were other parts of the system where rollback might be instigated that would be the right thing to do: those other parts of the system should not be aware of particulars of the disk layout.

However, since the chain database is *exclusively* in charge of switching to forks, all the logic can be isolated to the chain database. So, when we have a short volatile fragment, we will just not roll back more than the length of that fragment. Conceptually this can be justified also: the fact that I is the tip of the immutable DB means that *at some point* it was in our chain at least k blocks back, and so we considered it to be immutable: the fact that some data loss occurred does not really change that. We may still roll back more than k blocks when disk corruption occurs in the immutable database, of course.

One use case of the current fragment merits a closer examination. When the chain sync client (chapter 14) looks for an intersection between our chain and the chain of the upstream peer, it sends points from our chain fragment. If the volatile fragment is shorter than k due to data corruption, the client would have fewer points to send to the upstream node. However, this is the correct behaviour: it would mean we cannot connect to upstream nodes who fork more than k of what *used to be* our tip before the data corruption, even if that's not where our tip is anymore. In the extreme case, if the volatile database gets entirely erased, only a single point is available (the tip of the immutable database I), and hence we can only connect to upstream nodes that have I on their chain. This is precisely stating that we can only sync with upstream nodes that have a chain that extends our immutable chain.

12.5 Garbage collection

Blocks on chains that are never selected, or indeed blocks whose predecessor we never learn, will eventually be garbage collected when their slot number number is more than k away from the tip of the selected chain.[43]

> **Known bug**. The chain DB (more specifically, the volatile DB) can still grow without bound if we allow upstream nodes to rapidly switch between forks; this should be addressed at the network layer (for instance, by introducing rate limiting for rollback in the chain sync client, chapter 14).

[43] This is slot based rather than block based for historical reasons only; we should probably change this.

Although this is GC of the volatile DB, I feel it belongs here more than in the volatile DB chapter because here we know *when* we could GC. But perhaps it should be split into two: a section on how GC is implemented in the volatile DB chapter, and then a section here how it's used in the chain DB. References from elsewhere in the report to GC should probably refer here, though, not to the vol DB chapter.

12.5.1 GC delay

For performance reasons neither the immutable DB nor the volatile DB ever makes explicit `fsync` calls to flush data to disk. This means that when the node crashes, recently added blocks may be lost. When this happens in the volatile DB it's not a huge deal: when the node starts back up and the chain database is initialised we just run chain selection on whatever blocks still remain; in typical cases we just end up with a slightly shorter chain.

However, when this happens in the immutable database the impact may be larger. In particular, if we delete blocks from the volatile database as soon as we add them to the immutable database, then data loss in the immutable database would result in a gap between the volatile database and the immutable database, making *all* blocks in the volatile database unusable. We can recover from this, but it would result in a large rollback (in particular, one larger than k).

To avoid this, we currently have a delay between adding blocks to the immutable DB and removing them from the volatile DB (garbage collection). The delay is configurable, but should be set in such a way that the possibility that the block has not yet been written to disk at the time of garbage collection is minimised;a a relatively short delay should suffice (currently we use a delay of 1 minute), though there are other reasons for preferring a longer delay:

- Clock changes can more easily be accommodated with more overlap (section 24.11)
- The time delay also determines the worst-case validity of iterators (todo: reference to relevant section). TODO

Larger delays will of course result in more overlap between the two databases. During normal node operation this might not be much, but the overlap might be more significant during bulk syncing.

Notwithstanding the above discussion, an argument could be made that the additional complexity due to the delay is not worth it; even a "rollback" of more than k is easily recovered from[44], and clock changes as well, as iterators asking for blocks that now live on distant chains, are not important use cases. We could therefore decide to remove it altogether.

12.6 Resources

In the case of the chain DB, the allocation function will be wrapped in a `runWithTempRegistry` combinator, which will hold an empty resulting state. This is because as mentioned in 3.5.1, we only get values that do not leak implementation details and therefore we can't run any checks, but still we want to keep track of the resources. The allocation of each of the databases (Immutable DB and Volatile DB) will be executed using the combinator `runInnerWithTempRegistry` so that each of them performs the relevant checks on the `OpenState` they return but such checks are not visible (nor runnable) on the chain DB scope.

The threads that are spawned during the initialization of the database will be registered in the node general registry as they won't be directly tracked by the chain DB API but instead will coexist on its side.

The final step of ChainDB initialization is registering itself in the general registry so that it is closed in presence of an exception.

[44]Note that the node will never actually notice such a rollback; the node would crash when discovering data loss, and then restart with a smaller chain

Chapter 13

Mempool

Whenever a block producing node is the leader of a slot (section 4.2.4), it gets the chance to mint a block. For the Cardano blockchain to be useful, the minted block in the blockchain needs to contain *transactions*. The *mempool* is where we buffer transactions until we are able to mint a block containing those transactions.

Transactions created by the user using the wallet enter the Mempool via the local transaction submission protocol (see section 15.3). As not every user will be running a block producing node or stakepool, these transactions should be broadcast over the network so that other, block producing, nodes can include these transactions in their next block, in order for the transactions to ends up in the blockchain as soon as possible. This is accomplished by the node-to-node transaction submission protocol, which exchanges the transactions between the mempool of the nodes in the network.

link?

Naturally, we only want to put transactions in a block that are valid w.r.t. the ledger state against which the block will be applied. Putting invalid transactions in a block will result in an invalid block, which will be rejected by other nodes. Consequently, the block along with its rewards is lost. Even for a node that is not a block producer, there is no point in flooding the network with invalid transactions. For these reasons, we validate the transactions in the mempool w.r.t. the current ledger state and remove transactions that are no longer valid.

13.1 Consistency

Transactions themselves affect the ledger state, consequently, the order in which transactions are applied matters. For example, two transactions might try to consume the same UTxO entries. The first of the two transactions to be applied determines will be valid, the second will be invalid. Transactions can also depend on each other, hence the transactions that are depended upon should be applied first. Consequently, the mempool needs to decide how transactions are ordered.

We chose a simple approach: we maintain a list of transactions, ordered by the time at which they arrived. This has the following advantages:

- It's simple to implement and it's efficient. In particular, no search for a valid subset is ever required.
- When minting a block, we can simply take the longest possible prefix of transactions that fits in a block.
- It supports wallets that submit dependent transactions (where later transaction depends on outputs from earlier ones).

We call this *linear consistency*: transactions are ordered linearly and each transaction is valid w.r.t. the transactions before it and the ledger state against which the mempool was validated.

The mempool has a background thread that watches the current ledger state exposed by the Chain DB (chapter 12). Whenever it changes, the mempool will revalidate its contents w.r.t. that ledger state. This

ensures that we no longer keep broadcasting invalid transactions and that the next time we get to mint a block, we do not have to validate a bunch of invalid transactions, costing us more crucial time.

13.2 Caching

The mempool caches the ledger state resulting from applying all the transactions in the mempool to the current ledger state. This makes it quick and easy to validate incoming transactions, they can simply be validated against the cached ledger state without having to recompute it for each transaction. As discussed in section 10.1, the memory cost of this is minimal. When the incoming transaction is valid w.r.t. the cached ledger state, we append the transaction to the mempool and we cache the resulting ledger state. TODO

talk about the slot for which we produce

13.3 TxSeq

TODO

efficiently get the first x transactions that fit into the given size TODO

discuss `TicketNo`

13.4 Capacity

TODO

discuss dynamic capacity, based on twice the max block (body?) size in the protocol parameters in the ledger add transactions one-by-one for better concurrency and fewer revalidation in case of retries TODO

Part III

Mini protocols

Chapter 14

Chain sync client

14.1 Header validation

Discuss the fact that we validate headers (maybe a forward reference to the genesis chapter, where this becomes critical).

Discuss that this means we need efficient access to the k most recent ledger states (we refer to this section for that).

14.2 Forecasting requirements

Discuss that forecasting must have sufficient range to validate a chain longer than our own chain, so that we can meaningfully apply chain selection.

NOTE: Currently chapter 22 contains such a discussion.

14.3 Trimming

14.4 Interface to the block fetch logic

We should discuss here the (very subtle!) reasoning about how we establish the precondition that allows us to compare candidates (section 11.1.2). See `plausibleCandidateChain` in `NodeKernel` (PR #2735).

Chapter 15

Mini protocol servers

The division of work between the network layer and the consensus layer when it comes to the implementation of the clients and servers of the mini protocols is somewhat pragmatic. Servers and clients that do significant amounts of network layer logic (such as block fetch client which is making delta-Q related decisions, node-to-node transaction server and client, which are dealing with transaction windows, etc), live in the network layer. Clients and servers that primarily deal with consensus side concerns live in the consensus layer; the chain sync client (chapter 14), is the primary example of this. There are also a number of servers for the mini protocols that do little more than provide glue code between the mini protocol and the consensus interface; these servers are described in this chapter.

15.1 Local state query

15.2 Chain sync

15.3 Local transaction submission

Unlike remote (node to node) transaction submission, local (client to node) transaction submission does not deal with transaction windows, and is consequently much simpler; it therefore lives consensus side rather than network side.

15.4 Block fetch

Part IV

Hard Fork Combinator

Chapter 16

Overview

16.1 Introduction

We should discuss terminology here: what we mean by a hard fork, and how that is different from how the word is usually used.

We should mention that era transitions happen at epoch boundaries only.

Mention that we had to adjust the consensus layer in some ways:

- Simplified chain selection (tip only; section 4.1.1)
- Remove the assumption slot/time conversion is always possible (chapter 17)

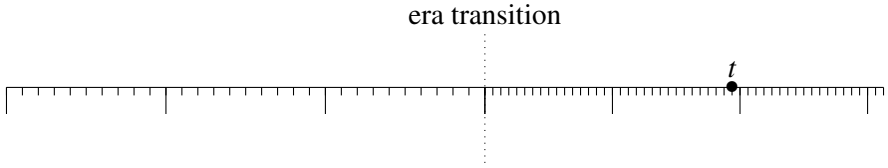
Chapter 17

Time

17.1 Introduction

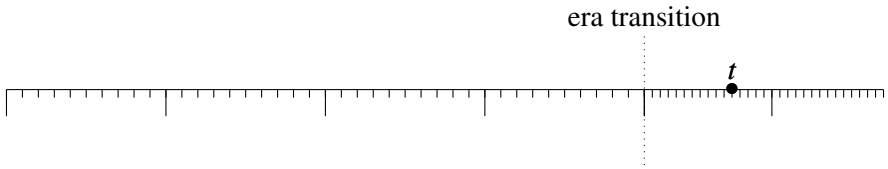
A fundamental property of the Ouroboros family of consensus protocols is that they all divide time into discrete chunks called *slots*; typically the duration of a slot is on the order of seconds. In most Ouroboros protocols slots are grouped into *epochs*, with certain changes to the consensus chain state happening at various points in an epoch. All nodes running the blockchain agree on a *system start time* (as a UTC time) through the chain's genesis configuration, making the translation from a particular wallclock time to a slot number easy: subtract the system start time from the wall clock time, and divide by the slot length. This assumption that the mapping between wall clock and slot or epoch numbers is always available permeated the consensus layer. Unfortunately, it is not a valid assumption in the presence of hard forks.

It's not difficult to illustrate this with an example. Suppose we want to know which slot time t corresponds to in:



We can read off from this depiction that t is in epoch 1 *of the second era*, and relative slot 14 within that epoch. Since there are 16 slots to an epoch in that era, that makes it slot $1 \times 16 + 14 = 30$ within that era. The second era was preceded by three epochs in the first era, each of which contained 10 slots, which means that time t was slot $3 \times 10 + 30 = 60$ globally.

But now consider how this calculation changes if the era transition would have happened one epoch later:



Slot t is now in epoch 0 of the second era, with relative slot 11, making it slot $0 \times 16 + 11 = 11$ within the second era. Since the second era got preceded by *four* epochs of the first era, that makes time t global slot $4 \times 10 + 11 = 51$.

All of this would be no more than a minor complication if the exact moment of the era transition would be statically known. This however is not the case: the moment of the era transition is decided *on the chain itself*. This leads to the inevitable conclusion that time/slot conversions depend on the ledger state, and may indeed be impossible: the slot at time t is *simply not yet known* if the transition to era 2 has not been decided yet.

17.2 Slots, blocks and stability

In section 4.1.2 we discussed the fundamental parameter k: blocks that are more than k blocks away from the tip of the chain are considered to be immutable by the consensus layer and no longer subject to rollback. We say that such blocks are *stable*.

The ledger layer itself also depends on stability; for example, in Shelley the stake distribution to be used for the leadership check needs to be stable before it is adopted (this avoids malicious nodes from inspecting the leadership schedule and then trying to cause a rollback if that leadership schedule is not beneficial to them).

The ledger layer however does not use block numbers to determine stability, but uses slot numbers to approximate it instead. This ultimately comes from the fact that in Ouroboros the length of an *epoch* is based on slots, not blocks, although this is something we may wish to revisit (section 24.4).

Depending on the particular choice of consensus algorithm, not all slots contain blocks. For example, in Praos only a relatively small percentage of slots contain blocks, depending on the Praos f parameter (in Shelley, f is set to 5%). However, the various Ouroboros protocols come with proofs (actually, a probabilistic argument) providing a window of a certain number of slots that is guaranteed to contain at least k blocks; for example, for Ouroboros Classic that window is $2k$ slots[45], and for Ouroboros Praos that window is $3k/f$. Stability requirements in the ledger then take the form "at least $3k/f$ slots must have passed" instead of "at least k blocks must have been applied".

17.3 Definitions

17.3.1 Time conversion

As we saw in section 17.1, we cannot do time conversions independent of a ledger state. This motivates the following definition:

Definition 17.1 (Time conversion)**.** Let $\texttt{Conv}_\sigma(t)$ be the function that converts time t, with t either specified as a wallclock time, a slot number, or an epoch number, to a triplet

$$\text{(wallclock time, slot number, epoch number)}$$

using ledger state σ, provided σ contains sufficient information to do so; $\texttt{Conv}_\sigma(t)$ is undefined otherwise.

Since all past era transitions are (obviously) known, time conversion should always be possible for points in the past. Let $\texttt{tip}(\sigma)$ be the (time of) the most recently applied block in σ. Then:

Property 17.1 (Conversion for past points)**.** $\texttt{Conv}_\sigma(t)$ should be defined for all $t \leq \texttt{tip}(\sigma)$.

Furthermore, we assume that time conversion is monotone:

Property 17.2 (Monotonicity of time conversion)**.** If $\texttt{Conv}_\sigma(t)$ is defined, then $\texttt{Conv}_{\texttt{apply}_{bs}(\sigma)}(t)$ must be as well and

$$\texttt{Conv}_{\texttt{apply}_{bs}(\sigma)}(t) = \texttt{Conv}_\sigma(t)$$

where $\texttt{apply}_{bs}(\sigma)$ denotes the ledger state after applying blocks bs.

17.3.2 Forecast range

Under certain conditions a ledger state may be usable to do time conversions for slots ahead of the ledger state.

Definition 17.2 (Forecast range)**.** We say that time $t > \texttt{tip}(\sigma)$ is within the forecast range of σ if $\texttt{Conv}_\sigma(t)$ is defined.

Note that monotonicity (property 17.2) should still apply.

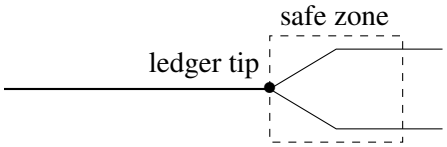
[45] Without much justification, we adopt this same window for PBFT as well. It is almost certainly a gross overestimation.

17.3.3 Safe zone

In order to be able to have a non-empty forecast range, we need to restrict when era transitions can occur.

Definition 17.3 (Safe zone)**.** A *safe zone* is a period of time ahead of a ledger's tip in which an era transition is guaranteed not to occur if it is not yet known.

Intuitively, a non-empty safe zone means that there will be time between an era transition being announced and it happening, no matter how the chain is extended (no matter which blocks are applied):



(17.1)

17.4 Ledger restrictions

17.4.1 Era transitions must be stable

Monotonicity (property 17.2) only talks about a chain's linear history; since the consensus layer needs to deal with rollbacks (switching to alternative chains) too, we will actually need a stronger property. Clearly, time conversions cannot be invariant under switching to arbitrary chains; after all, alternative chains might have era transitions in different places. The consensus layer however does not *support* switching to arbitrary alternative chains; we have a maximum rollback (section 4.1.2), and we never switch to a shorter chain (section 4.1.1, assumption 4.2). This means that we can model switching to an alternative chain as

$$\texttt{switch}_{(n,\, bs)}(\sigma)$$

where $n \leq k$ indicates how many blocks we want to rollback, bs is a list of new blocks we want to apply, and $\texttt{length}\ bs \geq n$.

Property 17.3 (Time conversions stable under chain evolution)**.** If $\texttt{Conv}_{\sigma}(t)$ is defined, then so is $\texttt{Conv}_{\texttt{switch}_{(n,\, bs)}(\sigma)}(t)$ and moreover

$$\texttt{Conv}_{\sigma}(t) = \texttt{Conv}_{\texttt{switch}_{(n,\, bs)}(\sigma)}(t)$$

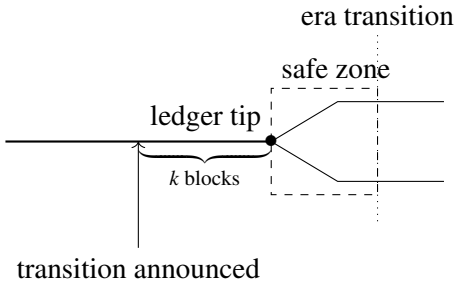
Intuitively, property 17.3 says that we might not be able to do time conversion for some time t because it's outside our current forecast range, but *if* it is within forecast range, then we don't need to earmark the answers we get from conversion as "subject to rollback": either we don't know, or we know for sure. This requirement may not be strictly *required* for consensus to operate (section 24.8), but it is a useful assumption which simplifies reasoning about time both within consensus and within clients of the consensus layer such as the wallet.

The existence of safe zones is not sufficient to establish this stronger property, in two ways:

- If we switch from a chain where an era transition is already known but far in the future, to a chain on which the era transition happens much sooner (or indeed, to a chain on which the era transition is not yet known), then the forecast range will shrink and hence $\texttt{Conv}_{\texttt{switch}_{(n,\, bs)}(\sigma)}(t)$ might not be defined, even if $\texttt{Conv}_{\sigma}(t)$ is.
- Conversely, if we switch from a chain on which the era transition is happening relatively soon, to a chain on which the era transition is happening later, then the forecast range will not shrink, but the time conversions on both chains will not agree with each other.[46]

[46] Going from a chain on which the era transition is not yet known to one in which it *is* known is not problematic, due to safe zones.

The key problem is that switching to an alternative chain can change our information about future era transitions, and hence result in different time conversions. We therefore insist that an era transition is not considered "known" until the block confirming the era transition is stable (no longer subject to rollback). This means that the minimum distance from the announcement of the era transition to the actual era transition must be long enough to guarantee k blocks plus the width of the safe zone:



(17.2)

How many slots are required to guarantee at least k blocks is consensus protocol specific; for example, for Praos this is typically set to $3k/f$ slots, where f is the active slot coefficient.

Many ledgers set the width of the safe zone such that it guarantees at least k blocks, but *in principle* there is no need for the width of the safe zone to be related to k at all, although other parts of consensus might have requirements for the width of the safe zone; we will discuss that in the next section (section 17.4.2).

17.4.2 Size of the safezones

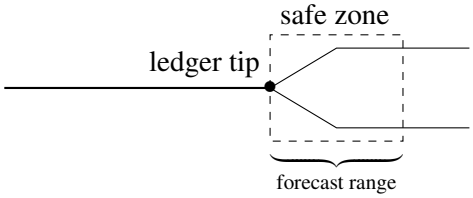
The most important example of where we might need to do time translation for blocks ahead of the ledger's tip is forecasting the Shelley ledger view (section 5.2). The Shelley ledger view contains an abstraction called `EpochInfo` allowing the ledger to do time conversions, for example to decide when rewards should be allocated.

As discussed in section 5.2.3, it is important that the forecast range of the ledger to allow us to validate at least $k + 1$ blocks after the ledger tip; consequently, the safe zone of the ledger must be wide enough to guarantee that it can span $k + 1$ blocks. This combination of the requirements of the ledger with the header/body split (section 3.1.1) means that in practice the width of the safe zone should be at least equal to the forecast range of the ledger, and hence defined in terms of k after all.

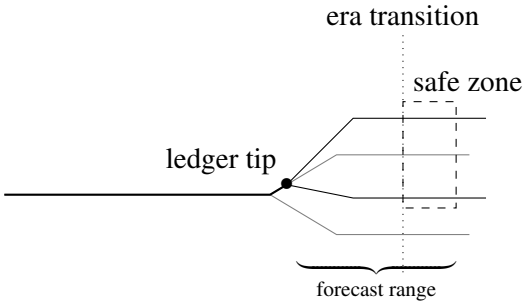
17.4.3 Stability should not be approximated

We discussed in section 17.2 that the ledger uses uses slot numbers to approximate stability. Such an approximation would violate property 17.3, however. Although we never switch to a shorter chain in terms of blocks, it is certainly possible that we might switch to a chain with a smaller *slot* number at its tip: this would happen whenever we switch to a longer but denser chain. If stability would be based on slot numbers, this might mean that we could go from a situation in which the era transition is considered known (and hence the forecast extends into the next era) to a situation in which the era transition is not yet considered known (and hence the forecast range only includes the safe zone in the current era).

Admittedly such a reduction of the forecast range would be temporary, and once the era transition is considered known again, it will be in the same location; after all, the block that confirmed the era transition *is* stable. This means that any previously executed time conversions would remain to be valid; however, the fact that the forecast range shrinks might lead to unexpected surprises. The consensus layer therefore does not use the ledger's layer notion of stability, but instead maintains additional state so that it can use *block* numbers rather than slot numbers to determine stability and be able to determine stability precisely rather than approximate it.



(17.4)



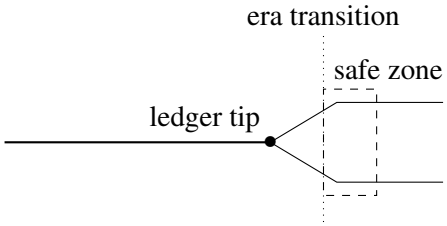
(17.5)

Figure 17.1: Era transition becoming known

17.5 Properties

17.5.1 Forecast ranges arising from safe zones

Slot length and epoch size can only change at era transitions. This means that if the transition to the next era is not yet known, any time t within the era's safe zone is guaranteed to be within the era's forecast range. If the transition to the next era *is* known, the safe zone of the current era is not relevant, but the safe zone of the next era is:



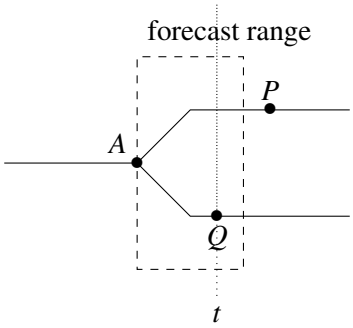
(17.3)

The safe zone of the next era might be smaller or larger than (or indeed of equal size as) the safe zone of the previous era; in this example it happens to be smaller.

Figure 17.1 shows how the forecast range changes as the next era transition becomes known; as shown, the next era starts at the earliest possible moment (right after the safe zone); in general it could start later than that, but of course not earlier (that would violate the definition of the safe zone).

17.5.2 Cross-fork conversions

Lemma 17.1 (Cross fork conversions)**.** Suppose we have the ledger state at some point P, and want to do time conversions for time t of a point Q on a different fork of the chain:



Provided that Q is within the forecast range of the common ancestor A of P and Q, the ledger state at P can be used to do time conversions for point t.

Proof. Since t is within the forecast range at A, by definition $\mathsf{Conv}_A(t)$ is defined. By monotonicity (property 17.2) we must have

$$\mathsf{Conv}_A(t) = \mathsf{Conv}_P(t)$$
$$\mathsf{Conv}_A(t) = \mathsf{Conv}_Q(t)$$

It follows that $\mathsf{Conv}_P(t) = \mathsf{Conv}_Q(t)$. □

17.6 Avoiding time

Time is complicated, and time conversions were pervasive throughout the consensus layer. Despite the exposition above and the increased understanding, we nonetheless have attempted to limit the use of time as much as possible, in an attempt to simplify reasoning whenever possible. The use of time within the core consensus layer is now very limited indeed:

1. When we check if we are a slot leader and need to produce a block, we need to know the current time as a slot number (We should discuss this somewhere. The chapter on the consensus protocol discusses the protocol side of things, but not the actual "fork block production" logic.) TODO.
2. When we add new blocks to the chain DB, we need to check if their slot number is ahead of the wallclock (section 11.5).
3. Specific consensus protocols may need to do time conversions; for example, Praos needs to know when various points in an epoch have been reached in order to update nonces, switch stake distribution, etc.

None of these use cases require either forecasting or cross-chain conversions. The most important example of where forecasting is required is in projecting the ledger view, as discussed in section 17.4.2. Cross-fork conversions (section 17.5.2) may arise for example when the consensus layer makes time conversions available to tooling such as the wallet, which may use it for example to show the wallclock of slots of blocks that may not necessarily live on the current chain.

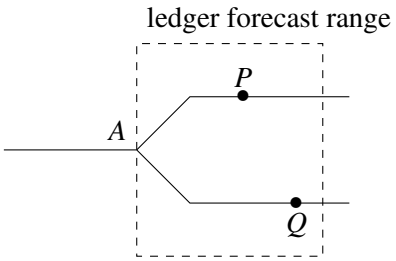
Keeping track of era transitions, and providing time conversions that take them into account, is the responsibility of the hard fork combinator and we will discuss it in more detail in section 18.2.

In the remainder of this section we will discuss some simplifications that reduced the reliance on time within the consensus layer.

17.6.1 "Header in future" check

Recall from section 3.1.1 that block downloading proceeds in two steps: first, the chain sync client downloads the block header and validates it; if it finds that the header is valid, the block download logic may decide to also download the block body, depending on chain selection (sections 4.1.1 and 4.2.1).

Suppose the node's own ledger state is at point P, and the incoming header is at point Q. In order to validate the header, we need a ledger *view* at point Q without having the ledger *state* at point Q; this means that point Q must be within the ledger's forecast range at the common ancestor A of P and Q (section 5.2):



As we have seen in section 17.5.2, if Q is within the *time* forecast range at A—put another way, if the time forecast range is at least as wide as the ledger forecast range—then we also can use the ledger state at P to do time conversions at point Q. Moreover, as we saw in section 17.4.2, for many ledgers that inclusion *must* hold. If we make this a requirement for *all* ledgers, in principle the chain sync client could do a header-in-future check.

For simplicity, however, we nonetheless omit the check. As we will see in the next section, the chain database must repeat this check *anyway*, and so doing it ahead of time in the chain sync client does not help very much; skipping it avoids one more use of time within the consensus layer. Indeed, a case could be made that we could skip header validation altogether, which would alleviate the need for forecasting *at all*; we will come back to this in section 24.6.

17.6.2 Ahead-of-time "block in future" check

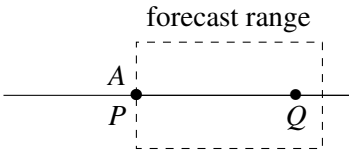
In the original design of the chain database, when a new block was added we first checked if the block's slot number was ahead of the wallclock, before considering it for chain selection. If it was ahead of the wallclock by a small amount (within the permissible clock skew), we then scheduled an action to reconsider the block when its slot arrived.

In order to compare the block's slot number to the wallclock, we can either convert the block's slot to a wallclock time, or convert the current wallclock time to a slot number. Both are problematic: the only ledger state we have available is our own current ledger state, which may not be usable to translate the current wallclock time to a slot number, and since we don't know anything about the provenance of the block (where the block came from), that ledger state may also not be usable to translate the block's slot number to a wallclock time. We now circumvent this problem by delaying the in-future check until we have validated the block,[47] and so can use the block's *own* ledger state to do the time conversion (section 11.5).

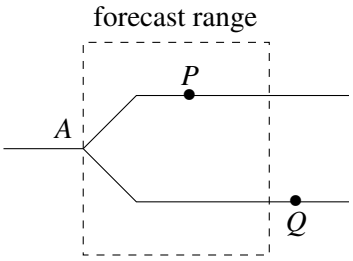
We saw in the previous section that the chain sync client *could* do the in-future check on headers, but the chain sync client is not the only way that blocks can be added to the chain database, so simply skipping the check in the chain database altogether, stipulating as a *precondition* that the block is not ahead of the wallclock, is not a good idea. Nonetheless it is worth considering if we could use a weaker precondition, merely requiring that the node's current ledger tip must be usable for time conversions for the slot number of the new block. Specifically, can we guarantee that we can satisfy this precondition in the chain sync client, if we do the in-future check on headers after all?

It turns out that in general we cannot, not even in relatively common cases. Consider again the diagram from section 17.6.1, but specialised to the typical case that the upstream node is on the same chain as we are, but a bit ahead of us:

[47] One might worry about vulnerability to a DoS attack, but the scope for an adversary to flood us with blocks from the future is limited, for two reasons. First, we still *validate* all headers, and the forecast range will determine quite how far we can look ahead. Second, the moment that the adversary feeds us with enough blocks from the future that their chain becomes longer than our own, we will download the corresponding blocks, find that they are from the future, mark them as invalid (provided it exceeds clock skew) and disconnect from the peer which we now consider to be potentially adversarial.



Since P and Q are on the same chain, point P is necessarily also the "intersection" point, and the distance between P and Q can only arise from the block download logic lagging behind the chain sync client. Now consider what happens when the node switches to an alternative fork:



Note what happens: since the node is switching to another fork, it must rollback some blocks and then roll forward; consequently, the intersection point A moves back, and P moves forward (albeit on a different chain). Q stays the same, *but might have fallen out of the forecast range at A*.

This means that even if the chain sync client was able to verify that a header (at point Q) was not ahead of the wallclock, if the node switches to a different fork before the block download logic has downloaded the corresponding block, when it presents that downloaded block to the chain database, the block might no longer be within the forecast range of the node's current ledger and the chain database will not be able to verify (ahead of time) whether or not the block is ahead of the wallclock. What's worse, unlike the chain sync client, the chain database has no access to the intersection point A; it all it has is the ledger's current tip at point P and the new block at point Q. It therefore has no reliable way of even determining *if* it can do time conversions for the new block.

17.6.3 "Immutable tip in future" check

The chain database never adopts blocks from the future (section 11.5). Nevertheless, it is possible that if the user sets their computer system clock back by (the equivalent of) more than k blocks, the immutable database (section 7.1) might contain blocks whose slot numbers are ahead of the wall clock. We cannot verify this during a regular integrity check of the immutable database because, as we have seen in this chapter, we would need a ledger state to do so, which we are not constructing during that integrity check. For now, we simply omit this check altogether, declaring it to be the user's responsibility instead to do a fresh install if they do reset their clock by this much.

However, in principle this check is not difficult: we initialise the immutable DB *without* doing the check, then initialise the ledger DB, passing it the immutable DB (which it needs to replay the most recent blocks, see chapter 10), and then ask the ledger DB for the ledger state corresponding to the tip of the immutable database. That ledger state will then allow us to do time conversions for any of the blocks in the immutable DB, trimming any blocks that are ahead of the wallclock.

17.6.4 Scheduling actions for slot changes

The consensus layer provides an abstraction called `BlockchainTime` that provides access to the current slot number. It also offers an interface for scheduling actions to be run on every slot change. However, if the node is still syncing with the chain, and does not have a recent ledger state available, the current slot number, and indeed the current slot length, are simply unknown: when the node uses its current ledger state to try and convert the current wallclock to a slot number, it will discover that the current wallclock is

outside the ledger's forecast range. In this case the blockchain time will report the current slot number as unavailable, and any scheduled actions will not be run.

We therefore limit the use of this scheduler to a single application only: it is used to trigger the leadership check (and corresponding block production, if we find we are a leader). This means that the leadership check will not be run if we are still syncing with the chain and have no recent ledger state available, but that is correct: producing blocks based on ancient ledger states is anyway not useful.

17.6.5 Switching on "deadline mode" in the network layer

Under normal circumstances, the priority of the network layer is to reduce *latency*: when a block is produced, it must be distributed across the network as soon as possible, so that the next slot leader can construct the *next* block as this block's successor; if the block arrives too late, the next slot leader will construct their block as the successor of the previous block instead, and the chain temporarily forks.

When we are far behind, however, the priority is not to reduce latency, but rather to improve *throughput*: we want to catch up as quickly as we can with the chain, and aren't producing blocks anyway (section 17.6.4).

In order to switch between these two modes we want to know if we are near the tip of the ledger—but how can we tell? If we know the current slot number (the slot number corresponding to the current wall clock), we can compare that current slot number to the slot number at the tip of the ledger. But, as we mentioned before, if we are far behind, the current slot number is simply unknown. Fortunately, we can use this to our advantage: if the slot number is unknown, we *must* be far behind, and hence we can use the decision, turning on deadline mode only if the slot number is known *and* within a certain distance from the ledger tip.

Chapter 18

Misc stuff. To clean up.

This is just a collection of random snippets right now.

18.1 Ledger

18.1.1 Invalid states

In a way, it is somewhat strange to have the hard fork mechanism be part of the Byron (section A.1) or Shelley ledger (section B.1) itself, rather than some overarching ledger on top. For Byron, a Byron ledger state where the *major* version is the (predetermined) moment of the hard fork is basically an invalid state, used only once to translate to a Shelley ledger. Similar, the *hard fork* part of the Shelley protocol version will never increase during Shelley's lifetime; the moment it *does* increase, that Shelley state will be translated to the (initial) state of the post-Shelley ledger.

This came from the Byron/Shelley appendix. Need to generalise a bit or provide context.

18.2 Keeping track of time

EpochInfo

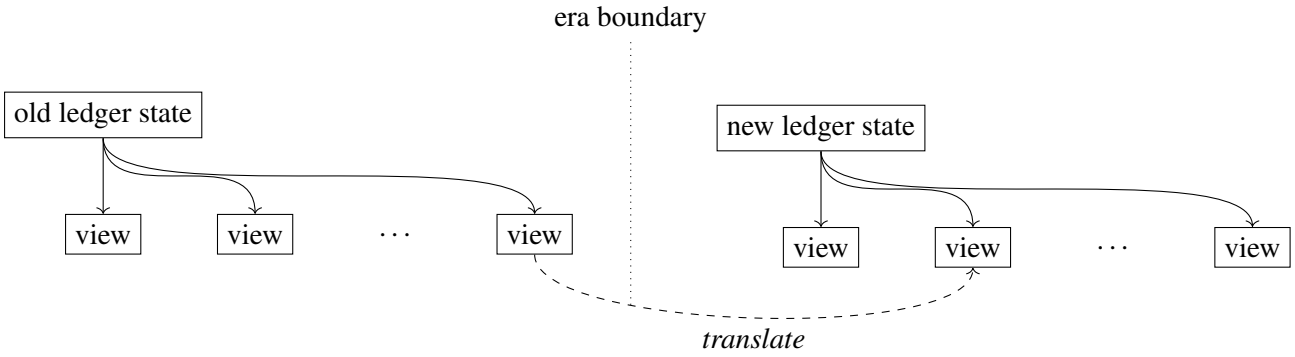
18.3 Failed attempts

18.3.1 Forecasting

As part of the integration of any ledger in the consensus layer (not HFC specific), we need a projection from the ledger *state* to the consensus protocol ledger *view* (section 5.1.4). As we have seen, the HFC additionally requires for each pair of consecutive eras a *state* translation functions as well as a *projection* from the state of the old era to the ledger view of the new era. These means that if we have $n + 1$ eras, we need n across-era projection functions, in addition to the $n + 1$ projections functions we already have *within* each era.

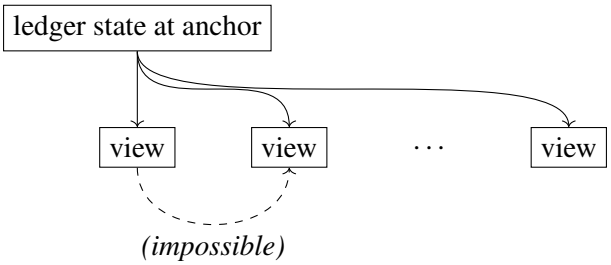
Once we write these sections, add back references here

This might feel a bit cumbersome; perhaps a more natural approach would be to only have within-era projection functions, but require a function to translate the ledger view (in addition to the ledger state) for each pair of eras. We initially tried this approach; when projecting from an era to the next, we would first ask the old era to give us the final ledger view in that era, and then translate this final ledger view across the eras:



The problem with this approach is that the ledger view only contains a small subset of the ledger state; the old ledger state might contain information about scheduled changes that should be taken into account when constructing the ledger view in the new era, but the final ledger view in the old era might not have that information.

Indeed, a moment's reflection reveals that this cannot be right the approach. After all, we cannot step the ledger state; the dashed arrow in



is not definable: scheduled changes are recorded in the ledger state, not in the ledger view. If we cannot even do this *within* an era, there is no reason to assume it would be possible *across* eras.

We cannot forecast directly from the old ledger state to the new era either: this would result in a ledger view from the old era in the new era, violating the invariant we discussed in section 18.1.1.

Both approaches—forecasting the final old ledger state and then translating, or forecasting directly across the era boundary and then translating—also suffer from another problem: neither approach would compute correct forecast bounds. Correct bounds depend on properties of both the old and the new ledger, as well as the distance of the old ledger state to that final ledger view. For example, if that final ledger view is right at the edge of the forecast range of the old ledger state, we should not be able to give a forecast in the new era at all.

Requiring a special forecasting function for each pair of eras of course in a way is cheating: it pushes the complexity of doing this forecasting to the specific ledgers that the HFC is instantiated at. As it turns out, however, this function tends to be easy to define for any pair of concrete ledgers; it's just hard to define in a completely general way.

Part V

Testing

Chapter 19

Reaching consensus

19.1 Dire-but-not-to-dire

We should mention the PBFT threshold here section 4.5.3.

Chapter 20

The storage layer

Part VI

Future Work

Chapter 21

Ouroboros Genesis

21.1 Introduction

21.1.1 Background: understanding the Longest Chain rule

Recall the Praos chain selection rule:

Definition 21.1 (Longest Chain Rule)**.** A candidate chain is preferred over our current chain if

1. it is longer than our chain, and
2. the intersection point is no more than k blocks away from our tip.

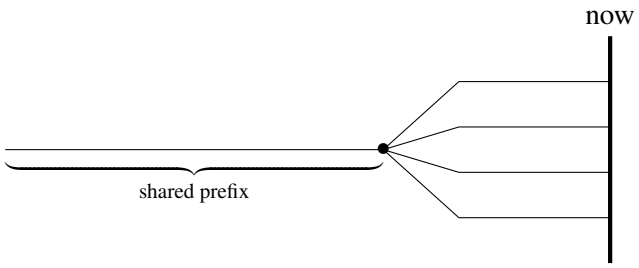
The purpose of chain selection is to resolve temporary forks that arise from the normal operation of the protocol (such as when there are multiple leaders in a single slot), and—importantly—to distinguish honest chains from chains forged by malicious nodes. It is not a priori clear why choosing longer chains over shorter chains would help distinguish malicious chains from honest chains: why would an honest chain be longer?

Recall that the leadership schedule is based on stake: a node's probability of being elected a leader in a given slot is proportional to their stake. By assumption, the malicious nodes in the system together have less stake than the honest nodes; security of the system as a whole critically depends on the presence of this honest majority. This means that when a malicious node extends the chain they can only produce a chain with relatively few filled slots: the honest chain will be *denser*. At least, this will be true near the intersection point: as we get further away from that intersection point, the malicious node can attempt to influence the leadership schedule for future slots to their advantage.

The Praos security analysis [6] tells us that provided all (honest) nodes are online all the time, they will all share the same chain, except for blocks near the tips of those chains. Moreover, blocks with a slot number ahead of the wall clock are considered invalid. This means that the only way[48] for one chain to be longer than another is by having more filled slots between the tip of the shared prefix and "now": in other words, they must be *denser*.[49]

[48]The chain sync client does actually allow for some clock skew. Headers that exceed the clock skew are however not included in chain selection.

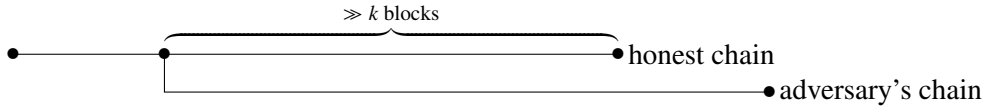
[49]A slightly subtle point arises from era transitions on the chain that may change parameters such as the active slot coefficient, allowing for denser chains after the transition. Although an adversary can introduce such a transition on their own fork, such a transition would not take effect until one stability window later, and so this won't affect the density near the intersection point. Era transitions that are agreed on on the main chain benefit the honest parties and the adversary equally.



This motivates the first part of the Longest Chain rule: chain length is a useful proxy for chain density, which in turn reflects honesty. The second part of the rule—the intersection point is no more than k blocks away from our tip—is important because we can only meaningfully compare density *near the intersection point*. As we get further away from the intersection point, an adversary can start to influence the leadership schedule. This means that if the adversary's chain forks off from the honest chain far back enough, they can construct a chain that is longer than the honest chain. The Longest Chain rule therefore restricts rollback, so that we will simply not even consider chains that fork off that far back. We can still resolve minor forks that happen in the honest chain during the normal operation of the protocol, because—so the analysis guarantees—those will not be deeper than k blocks.

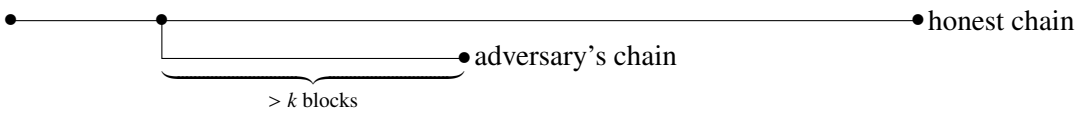
21.1.2 Nodes joining late

When new nodes join the network (or rejoin after having been offline for a while), they don't have the advantage of having been online since the start of the system, and have no sufficiently long prefix of the honest chain available. As we saw at the end of section 21.1.1, simply looking at chain length is insufficient to distinguish the honest chain from malicious chains: given enough time, an adversary can produce a chain that is longer than the honest chain:



When a node's current chain is somewhere along that common prefix and uses the longest chain rule, they will choose the adversary's chain rather than the honest chain. Moreover, they will now be unable to switch to the honest chain, because the intersection point with that chain is more than k blocks ago. If a node would get a "leg up" in the form of a reliable message telling which chain to adopt when joining the network (such a message is known as a "checkpoint" in the consensus literature), the Praos rule from that point forward would prevent them from (permanently) adopting the wrong chain, but Praos cannot be used to help nodes "bootstrap" when they are behind.

So far we have just been discussing Praos as it is described in theory. The situation in practice is worse. In the abstract models of the consensus algorithm, it is assumed entire chains are being broadcast and validated. In reality, chains are downloaded and validated one block at a time. We therefore don't see a candidate chain's *true* length; instead, the length of a candidate we see depends on how much of that candidate's chain we have downloaded[50]. Defining chain selection in terms of chain length, where our *perceived* chain length depends on what we decide to download, is obviously rather circular. In terms of the above discussion, it means that the adversary's chain doesn't even need to be longer than the honest chain:



[50]Even if nodes did report their "true length" we would have no way of verifying this information until we have seen the entire chain, so we can make no use of this information for the purpose of chain selection.

If the adversary's chain contains more than k blocks after the intersection point, and we *happen* to download that chain first, we would adopt it and subsequently be unable to switch to the honest chain; after all, that would involve a rollback of more than k blocks, which the Praos rule forbids.

21.1.3 The Density rule

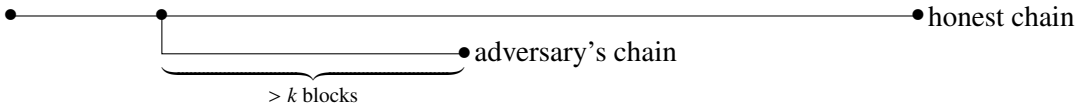
It is therefore clear that we need a different chain selection rule so that nodes can (re)join, and the Ouroboros Genesis paper [1] proposes one, shown in fig. 21.1. In this chapter we will work with a slightly simplified (though equivalent) form of this rule, which we will term the Density Rule:

Definition 21.2 (Density Rule)**.** A candidate chain is preferred over our current chain if it is denser (contains more blocks) in a window of s slots anchored at the intersection between the two chains.

(We will briefly discuss the differences between the rule in the paper and this one in section 21.2.3.) Technically speaking, s is a parameter of the rule, but the following default is a suitable choice both from a chain security perspective and from an implementation perspective:[51]

Definition 21.3 (Genesis window size)**.** The genesis window size s will be set to $s = 3k/f$.

Unlike the Longest Chain rule, the Density rule does not impose a maximum rollback. It does not need to, as it always considers density *at the intersection point*. This means that in a situation such as



if we happen to see and adopt the adversary's chain first, we can still adopt the honest chain (which will be denser at the intersection point, because of the honest majority).This would however involve a rollback of more than k blocks; we will discuss how we can avoid such long rollbacks in section 21.5.

21.2 Properties of the Density rule

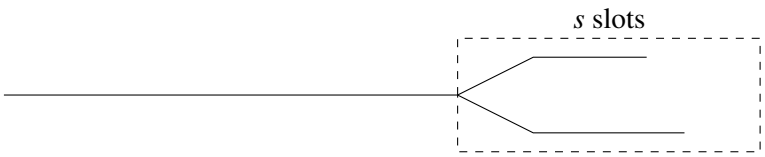
In this section we will study the Density rule and prove some of its properties. The improved understanding of the rule will be beneficial in the remainder of this chapter.

21.2.1 Equivalence to the Longest Chain rule

For nodes that are up to date, the Density rule rule does not change how chain selection works.

Lemma 21.1. When comparing two chains with an intersection that is at most s slots away from the two tips, the Density rule just prefers the longer chain.

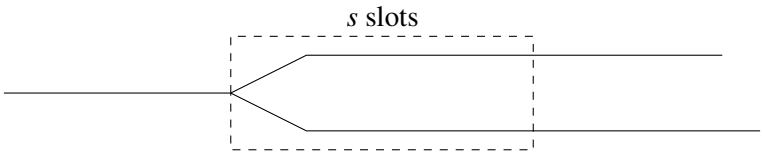
Proof. The two chains share a common prefix, and differ only in the blocks within the last s slots:



Since the chain length in this case is simply the length of the length of the common prefix plus the number of blocks in the window (i.e., their density), the longer chain will also be denser in the window.

Just to be very explicit, lemma 21.1 does *not* hold when the intersection is more than s slots away:

[51] If we change the epoch size, this value might have to be reconsidered, along with the ledger's stability window.



In this case of course the longer chain may well not be denser in the window. □

Lemma 21.2 (Rule Equivalence)**.** When comparing two chains with an intersection that is at most k blocks away from the two tips, the Density rule and the Longest Chain rule are equivalent.

Proof. First, observe that since the intersection is at most k blocks away, the maximum rollback condition of the Longest Chain rule is trivially satisfied. Remains to show that the intersection is at most s slots away, so that we can apply lemma 21.1. This is easiest to see by contradiction: suppose the intersection is *more* than s slots away. Then we would have a section on the chain which is more than s slots long but contains fewer than k blocks; the Chain Growth analysis in [6, 1] tells us that the probability of this happening is negligibly small (provided s is at least $3k/f$). □

Lemma 21.2 has a corollary that is of practical importance for the consensus layer, as it re-establishes an invariant that we rely on (assumption 4.2):

Lemma 21.3. Alert nodes (that is, an honest node that has consistently been online) will never have to switch to a shorter chain.

Proof. The Ouroboros Genesis analysis [1] tells us that alert nodes will never have to roll back by more than k blocks. In other words, the intersection between their current chain and any chain they might have to switch to will be at most k blocks ago. The lemma now follows from lemma 21.2. □

21.2.2 Honest versus adversarial blocks

In this section we will consider what kinds of chains an adversary might try to construct.

Lemma 21.4. An adversary cannot forge a chain that forks off more than k blocks from an alert node's tip and is denser than the alert node's chain at the intersection between the two chains.

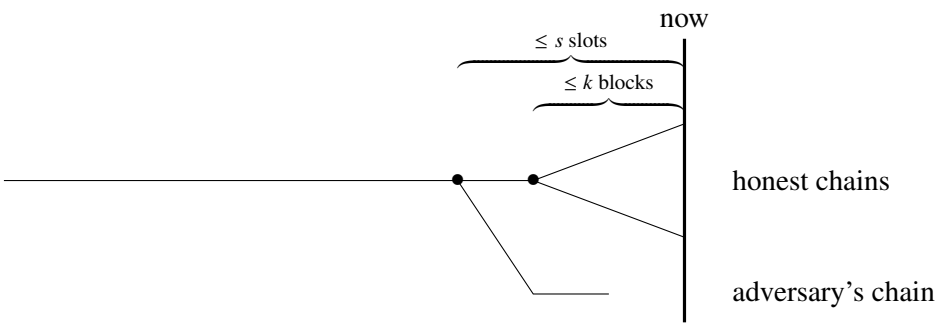
Proof. This is an easy corollary of the Ouroboros Genesis analysis. If the adversary would be able to construct such a chain, then by the Density rule the node should switch to it. As mentioned above, however, the analysis tells us that alert nodes never have to roll back more than k blocks. □

Lemma 21.4 has a useful specialisation for chains that an adversary might try to construct near the wallclock:

Lemma 21.5. An adversary cannot forge a chain that satisfies all of the below:

- It forks off no more than s slots from the wallclock.
- If forks off more than k blocks before the tip of an alert node.
- It is longer than the alert node's current chain.

Proof. This example satisfies the first two criteria but not the third:



The intersection between the alert node's chain and the adversarial chain is within s slots from the wallclock. This means that density at the intersection point is just chain length (lemma 21.1), and hence the property follows from lemma 21.4. □

21.2.3 The original genesis rule

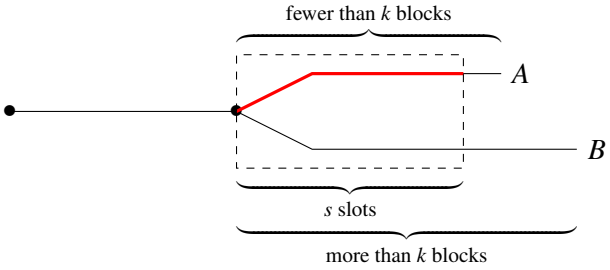
The Density rule is simplification of the rule as presented in the paper [1]. The original rule is shown in fig. 21.1, and paraphrased below:

Definition 21.4 (Genesis chain selection rule, original version)**.** A candidate chain is preferred over our current chain if

- The intersection between the candidate chain and our chain is **no more than** k blocks back, and the candidate chain is strictly **longer** than our chain.
- If the intersection *is* **more than** k blocks back, and the candidate chain is **denser** (contains more blocks) than our chain in a region of s slots starting at the intersection.

As we saw in lemma 21.2, the Density rule is equivalent to the Longest Chain rule if the intersection is within k blocks, so the original rule and the simplified form are in fact equivalent.

For completeness sake, we should note that this equivalence only holds for suitable choice of s. If s is much smaller (for example, the paper uses $s = \frac{1}{4}(k/f)$ in some places), then we might have a situation such as the following, where we have two chains A and B; A is denser than B at the intersection with B, but B is longer:



In this case, the original rule ends up preferring either A or B depending on the order in which we consider them, whereas the Density Rule would simply pick A.

21.3 Fragment selection

While the algorithms described in the literature on Ouroboros compare entire chains, we will want to compare *fragments* of chains: if at all possible we would prefer not to have to download and verify entire chains before we can make any decisions. As we have discussed in section 21.1.2, comparing fragments (prefixes) of chains using the Longest Chain rule does not actually make much sense, but in this section we will see that the situation is fortunately much better when we use the Density rule.

Definition 21.5 (Preferred fragment)**.** Let $\mathcal{S}$ be set of chain fragments, all anchored at the same point (that is, the fragments share a common ancestor), corresponding to some set of chains C. Then A is a preferred fragment in $\mathcal{S}$ if and only if A is a fragment of a preferred chain in C.

We will now establish the (necessary and sufficient) condition for fragment preference to be decidable. First, if we have to choose between two chains, we must see enough of those chains to do a density comparison.

Definition 21.6 (Known density)**.** We say that a chain fragment has a *known density* at some point p if either of the following conditions hold:

Parameters

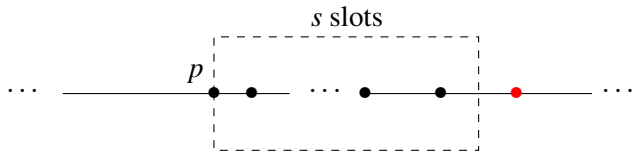
C_{loc}	Current chain
$\mathcal{N} = \{C_1, \ldots, C_M\}$	All possible chains (including our own)
k	Security parameter (section 4.1.2)
s	Genesis window size (Genesis rule specific parameter)
f	Active slot coefficient (section 4.6.1)

Algorithm

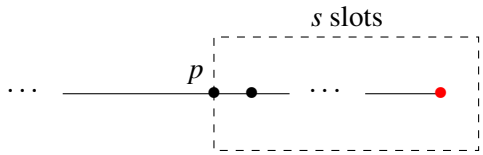
```
// Compare C_max to each C_i ∈ 𝒩
Set C_max ← C_loc
for i = 1 to M do
  if (C_i forks from C_max at most k blocks) then
    if |C_i| > |C_max| then // Condition A
      Set C_max ← C_i.
  else
    Let j ← max{ j' ≥ 0 | C_max and C_i have the same block in sl_j' }
    if |C_i[0 : j + s]| > |C_max[0 : j + s]| then // Condition B
      Set C_max ← C_i.
return C_max
```

Figure 21.1: Algorithm `maxvalid-bg`

1. The fragment contains a block after at least s slots:



2. The chain (not just the fragment[52]) terminates within the window:



Definition 21.7 (Look-ahead closure)**.** Let $\mathcal{S}$ be a set of chain fragments all anchored at the same point. We say that $\mathcal{S}$ is *look-ahead closed* if whenever there are two fragments $A, B \in \mathcal{S}$, the densities of A and B are known at their intersection.

Lemma 21.6 (Look-ahead closure is sufficient for fragment selection)**.** Let $\mathcal{S}$ be a look-ahead closed set of chain fragments. Then we can always choose a preferred fragment in $\mathcal{S}$.

Proof (sketch). In order to be able to pick a chain, we need to resolve forks. In order to resolve forks using the Density Rule, we need to know the density, but that is precisely what is guaranteed by look-ahead closure. □

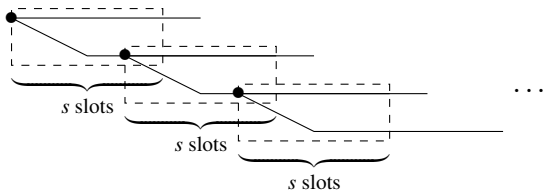
Lemma 21.6 is relevant because it reflects how the consensus layer uses chain selection:

1. We maintain a fragment of the chain for each upstream peer we track (chapter 14). The block fetch client (section 14.4) then picks a preferred fragment and downloads that.

[52] We can distinguish between these two cases because nodes report the tip of their chain as part of the chain sync protocol, independent from the headers that we have downloaded from those nodes.

2. When the chain database needs to construct the current chain (chapter 11), it constructs a set of chain fragments through the volatile DB, all anchored at the tip of the immutable database, picks a preferred fragment, and adopts that as the node's current chain.

However, lemma 21.6 is less useful than it might seem: the look-ahead closure requirement means that in the worst case, we still need to see entire chains before we can make a decision: every new intersection point requires us to see s more slots:

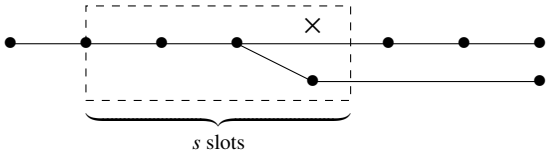


Moreover, due to the header/body split (section 3.1.1), when we are tracking the headers from an upstream peer, we cannot (easily) verify headers that are more than $3k/f$ slots away from the intersection between our chain and their chain (see also chapter 22). In the next section we will therefore consider how we can drop the look-ahead closure requirement.

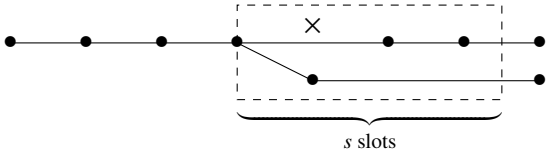
In case it is not obvious why we must only compare density at intersection points, in the remainder of this section we will consider an example that will hopefully clarify it. Suppose a malicious node with some stake intentionally skips their slot, after which the chain continues to grow as normal:



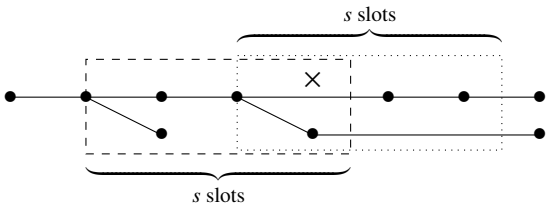
It is now trivial for the adversary to create an alternative chain that *does* have a block in that slot; if other nodes switch to the denser chain the moment they see a window of s slots that is denser, they would adopt the adversary's chain; after all, it has one more block in the window than the honest chain does:



Instead, we must wait until we make such a comparison until we have reached the intersection point:



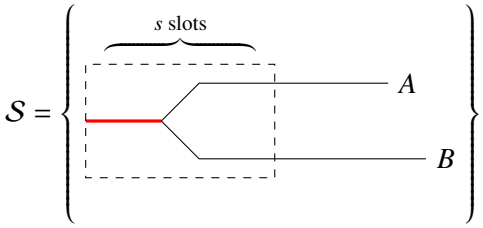
Since the adversary does not have sufficient stake, their chain will be less dense and we will therefore not select it. If the adversary creates another fork earlier on the chain, then we will resolve that fork when we encounter it using a window of s slots *anchored at that fork*, and then later resolve the second fork using a *different* window of s slots, anchored at the second fork:



21.4 Prefix selection

21.4.1 Preferred prefix

When a set $\mathcal{S}$ of chain fragments is not look-ahead closed, we may not be able to pick a best fragment. For example, in



we cannot choose between A and B; what we *can* say however is that no matter which of A and B turns out to be the better fragment, the common prefix of A and B (shown in red) will definitely be a prefix of that fragment. This example provides the intuition for the definition of a preferred prefix:

Definition 21.8 (Preferred prefix)**.** Given a set $\mathcal{S}$ of chain fragments, all anchored at the same point, a preferred prefix is a prefix Π of one of the fragments in $\mathcal{S}$, such that Π is guaranteed to be a prefix of a preferred fragment in the lookahead-closure of $\mathcal{S}$.

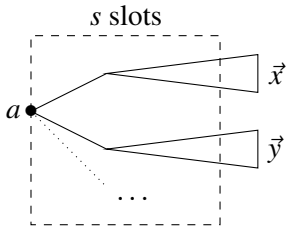
In other words, we may not be able to pick the best fragment out of $\mathcal{S}$, but we *can* pick a prefix which is guaranteed to be a prefix of whatever turns out to be the best fragment. Obviously, the empty fragment is always a valid choice, albeit not a particularly helpful one. Ideally, we would choose the *longest* preferred prefix. Fortunately, constructing such a prefix is not difficult.

21.4.2 Algorithm

We will now consider how we can choose the longest preferred prefix.

Definition 21.9 (Prefix selection)**.** Let $\mathcal{S}$ be a set of chain fragments all anchored at the same point a, such that all fragments have known density at point a. Then we can construct the longest preferred prefix in two steps:

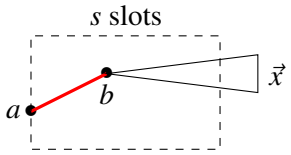
1. *Resolve initial fork.* Suppose $\mathcal{S}$ looks like this:



   where (without loss of generality) the common prefixes are non-empty. Suppose one of the $\vec{x}$ has the highest density[53] at point a; let's call it x_i. That means if we ever were to adopt any of the $\vec{y}, \ldots$, and then compared our chain to x_i, we would find that x_i is denser at the intersection point (which is precisely what we are comparing in this window here), and therefore switch to it. This means we can focus our attention on the $\vec{x}$. (Figure 21.2 justifies the greedy nature of this algorithm in a bit more depth.)

[53] If two fragments in different forks have have exactly the same density, we need a tie-breaker in order to be able to make progress. The genesis paper does not prefer either chain in such a scenario, switching only if another chain is strictly denser. We can therefore follow suit, and just focus on one of the two chains arbitrarily.

2. *Adopt common prefix.* Most of the time, the density of the $\vec{x}$ will yet not be known at point b. This means we do not yet know which x_i will turn out to be the best, but we *do* know that whichever it turns out to be, it will have the common prefix from a to b, so we choose this as the longest preferred prefix:



   One useful special case is when all of the $\vec{x}$ terminate within the window. In this case, their density *is* known, and we can just pick the densest fragment as the preferred prefix; the preferred prefix is then in fact the preferred fragment.

Prefix selection fits very naturally with the needs of the consensus layer (*cf.* the description of how consensus uses fragment selection in section 21.3):

1. When blockfetch applies prefix selection to the set of fragments of the upstream peers we are tracking, the computed prefix is precisely the set of blocks that blockfetch should download.
2. When the chain database applies prefix selection to the set of fragments through the volatile database, the computed prefix is precisely the chain that we should adopt as the node's current chain.

Lemma 21.7. If the intersection point between the chains of our upstream peers is at most k blocks away from their tips, prefix selection will choose the longest chain.

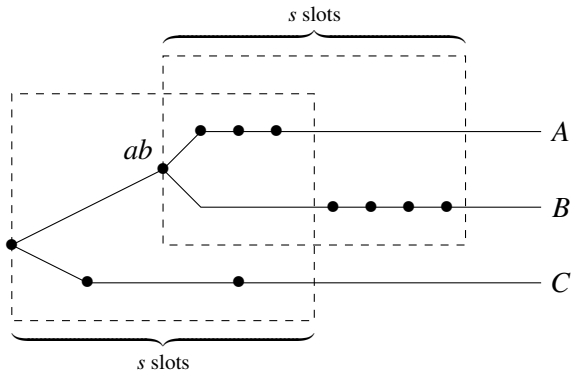
Proof (sketch). The proof is very similar to the proof of lemma 21.2. If the intersection is at most k blocks away, it will be less than s slots away; therefore prefix selection will be able to see the chains to their tip, the density of all chains will be known, and the densest fragment equals the longest one. □

21.4.3 Prefix selection on headers

When the chain sync client is deciding which of the chains of its upstream peers to download, it does so based on chains of *headers*. It does not download block bodies and so cannot verify them. As such, it is basing its decisions based on header validity *only*. However, this is then only used to tell the block fetch client which blocks to *download* (section 14.4); it does not necessarily mean that those blocks will be *adopted*. When the chain database performs chain selection (chapter 11), it will verify the blocks and discard any that turn out to be invalid. If any blocks *are* invalid, then the chain sync client will disconnect from the nodes that provided them, which in turn may change which prefix is chosen by prefix selection.

Indeed, the only reason to even validate headers at all is to avoid a denial of service attack where an adversary might cause us to waste resources. It may be worth reconsidering this risk, and balancing it against the costs for the implementation; the only reason we need forecasting, and a special treatment of low density chains (chapter 22) is that we insist we want to validate headers independent from blocks.

Step 1 in definition 21.9 makes greedy decisions, discarding the less dense chains at every step without looking further ahead. To justify this, it is instructive to consider an apparent counter-example to the validity of this strategy. Consider three chains A, B, C in the following example:



Depending on the order in which we consider the chains, we might pick any of these three chains:

order	**selected chain**
B, C, A	A
C, A, B	B
A, B, C	C

In practice this situation cannot occur, however. The Genesis analysis tells us that there will be single chain (the honest chain) which will be denser than any other chain (within the genesis window) at every intersection point. This is precisely what justifies the greedy nature of the algorithm: it will always hone in on the honest chain. In the counter-example above, there is no single chain with this property, and hence cannot arise.

Figure 21.2: Greedy chain selection

21.4.4 Known density

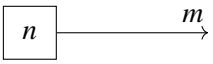
Definition 21.9 requires known density at the anchor a of the set, so that it can resolve the initial fork. This means we have to wait until we have downloaded enough headers from each peer: the last header we downloaded must either be outside the s window, or else it must be the last header on that peer's chain (in which case the peer will tell us so). If a peer tells us they have more headers but then do not provide them, we should disconnect from them after some time-out to avoid a potential denial of service attack.

If that header is outside the genesis window, we may not be able to validate it: since it is more than s slots away from the anchor point (and we only have a ledger state at the anchor point), it falls outside the ledger's forecast range. However, this does not matter: the presence of this header only tells us that we have seen everything we need to see to compute the density within the window; an invalid header after that window cannot increase the density *within* the window.

We can make one useful exception to the known density requirement: if there *is* no initial fork, we do not need to know the density at all and can go straight to step (2). This will allow a node to sync faster: consider a new node that is joining the network. In most cases, all of the node's upstream peers will report the *same* blocks for all but the last few blocks on the chain. Since there is no fork to resolve, we can start adopting each block the moment it is reported by all peers.

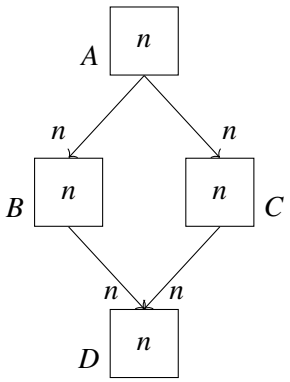
21.4.5 Propagation delays

Propagation delays will be a bit slower than before. To see why, let's consider the most ideal situation where all nodes are on exactly the same chain, and the next slot leader produces a new block. Let



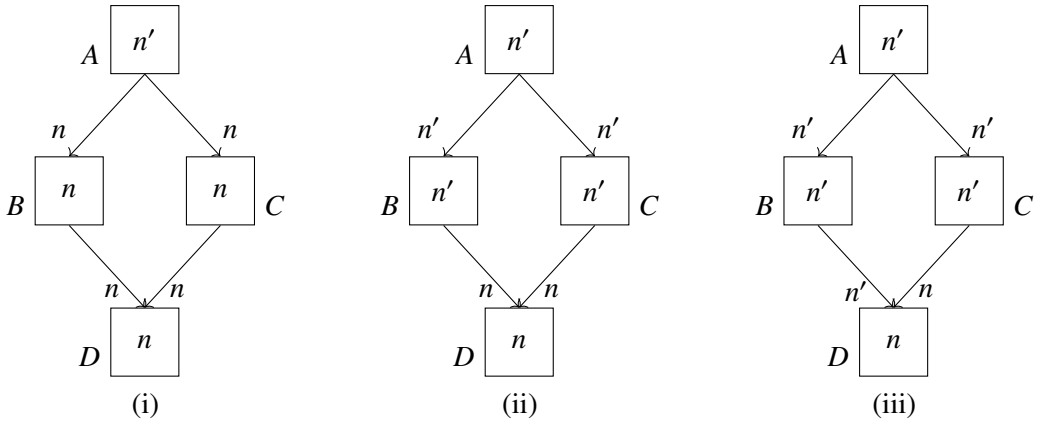
denote an upstream peer and a downstream peer. The upstream peer's tip is n, and is sending headers to the downstream peer; the most recent header that the downstream peer validated is m. When $n = m$, the downstream peer has "seen" the upstream peer's entire chain; in the terminology of this chapter, the density of the upstream peer is then *known*.

Consider a diamond topology with four peers $A \ldots D$, nodes B and C following node A, and node D following nodes B and C; suppose all nodes are up to date:



Now consider what happens when node A produces a new block:[54]

[54] Exact details of the chain sync protocol may mean that this example as it stands is not quite possible: the only way for D to become aware of C's new tip is when it receives the header. However, the point still remains; have A switch to a fork instead and repeat the reasoning.



Initially node A announces its header to B and C (i), which download it and verify it. Since their only peer is A, the density of "all" peers is now known and so both B and C adopt this new block and make it available to D (ii).

Suppose node D now downloads that header from B and verifies it (iii). Prior to the implementation of genesis, the chain selection rule would have concluded that this chain is longer, therefore preferred, and node D would be able to adopt the block before interacting with C.

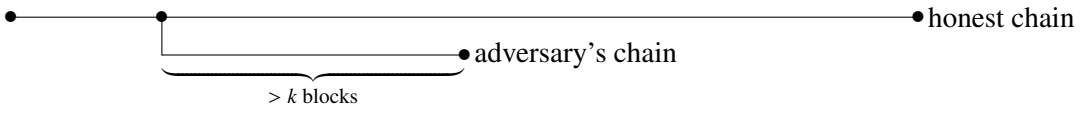
Prefix selection however will *wait*: it hasn't yet verified the header from C, and so it considers the density of C's chain's to be unknown, and hence it cannot make a decision. *This is the correct behaviour*: the old implementation would select C's chain immediately, but it would do so for a terrible reason: sure, it's longer, but only *because we downloaded more of that chain*; this is the circular reasoning we described above in section 21.1.2.

In principle it would be possible for D to *skip* the verification of any number of headers if those headers have already been verified from another peer. However, this would be optimising the code for the best case, with performance degrading when the peers are on different chains; this is something we avoid throughout (section 3.4).

21.5 Avoiding long rollbacks

21.5.1 Representative sample

At the end of section 21.1.3 we mentioned that if we have a situation such as



and we happen to adopt the adversary's chain first and later compare it to the honest chain, the Density Rule will prefer the honest chain and so we should switch, at the cost of a rollback of more than k blocks.

Such long rollbacks are problematic; we depend on having a maximum rollback of k blocks in many ways (sections 4.1.2, 7.1, 14.1 and 14.2 and others), and we will want to *continue* to depend on it. However, we needed this long rollback in this example only because we *first* adopted the adversary's chain, and *then* compared it to the honest chain. In the consensus layer even as it is today, we already don't do chain selection in such a chain-by-chain manner; as we saw in section 21.3 and again in section 21.4, we instead pick the best out of all currently known candidates. This means that as long as we are *aware* of both chains before we adopt either one, we will just pick the honest chain straight away, and we avoid the rollback.

This then is the solution to avoiding longer-than-k rollbacks: as long as we are not yet up to date with the main chain, we must make sure that we connect to a representative sample N_{rs} of upstream peers that the probability that *none* of them will serve us the honest chain is negligible (presumably aided by a probabilistic way of choosing upstream peers in the network layer), and avoid doing prefix selection until we have reached this threshold.

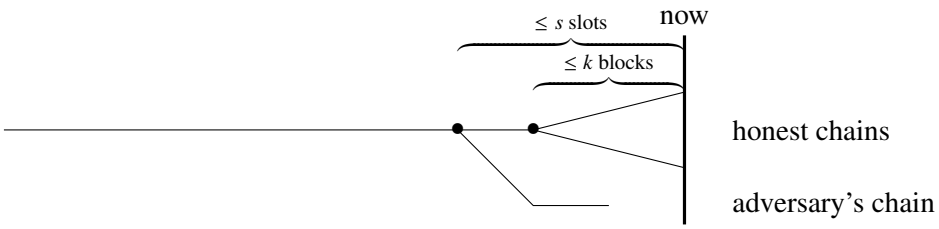
Of course, the need to avoid being eclipsed is not unique to genesis. When we are up to date we can however tolerate relatively long periods in which we do not see the honest chain; an exact analysis of the relation between the length of such an eclipse and the maximum chain divergence we can handle is beyond the scope of this chapter, but it seems safe to say that being eclipsed for short periods of time should be unproblematic. If however we are behind and are catching up with the chain, being eclipsed for even just one minute is already problematic: an attacker can easily feed us k blocks in that period, at which point we would be unable to switch to the honest chain.

21.5.2 Becoming alert

The only remaining decision to make is when we can *drop* this requirement: at which point is the state of the node comparable to the state of an alert node (a node that has consistently been online)?

Every block that we adopt through prefix selection that is more than s slots away from the wall clock must be a block on the honest chain, because at every fork we will only adopt blocks from the denser fragment. Moreover, all honest parties (all alert nodes) will agree on blocks that are more than s slots away from their tip. This means that we will not need to roll back at all: any block that we adopt which is more than s slots away from the wall-clock is a block that we can be sure about.

It is tempting to conclude from lemma 21.5 that as soon we have reached s slots from the wallclock, we can drop the requirement to be connected to N_{rs} nodes and restrict rollback to k blocks. This is however not the case. Recall what the situation looks like:



Lemma 21.5 tells us that the adversary cannot construct a chain that forks off more than k blocks from an alert node's chain and is longer than that chain. It does *not* tell us that it cannot contain more than k blocks after the intersection.[55] This means that if we dropped the requirement that we see all chains and then see and adopt the adversary's chain, we would be stuck, as switching to the honest chain would involve a rollback of more than k blocks.

Lemma 21.5 would only help us if we are somehow guaranteed that we have adopted one of the alert nodes' chains. But how can we tell? Fortunately, here we have a rare example of reality serendipitously lining up with theory. In the theoretical model, nodes collect entire chains broadcast by their peers, and then use the density rule to select the best one. Once they have done that, they have selected the same chain that any of the alert nodes might have selected, and so from this point forward their state is effectively indistinguishable from the state of an alert node.

In reality of course we cannot collect entire chains, and so we introduced the concept of prefix selection in order to be able to apply the Density rule incrementally. However, notice what happens when we have reached s slots from the wallclock: once we have filled our look-ahead window, *we will have seen every chain to its very tip*. Every fragment terminates within the window, which means that prefix selection will not just pick the preferred *prefix*, and not just the preferred *fragment*, but in fact the preferred *chain*. This means that just like the theory assumes, we have selected the best chain out of all possible chains, which means we can conclude we are now completely up to date and can resume normal operation.

Definition 21.10 (Recovering "alert" status)**.** When prefix selection can see all available chains to their tip, and we have selected and adopted the best one, the node is up to date.

(TODO: Christian tells me that the genesis proof also depends on such a "final chain selection". Would be good to refer to that, but I'm not sure where that happens in the paper.) TODO

[55] A worst-case adversary with near 50% stake would be able to construct $0.5 \times 3k = 1.5k$ blocks in $3k/f$ slots. The reasoning would be simpler if the adversary has at most 33% stake, as then they could indeed only construct k blocks in $3k/f$ slots.

21.5.3 Avoiding DoS attacks

Malicious nodes cannot abuse definition 21.10 in an attempt to prevent us from concluding we are up to date: as we saw, all blocks that get us to within s slots from the wallclock come from the honest chain, and once we have reached s slots from the wallclock, an adversary cannot present us with a chain that exceeds the window, since blocks with a slot number after the wall-clock are invalid.

We do have to be careful if we allow for clock skew however: if a malicious node presents us with a header past the s window (and hence past the wallclock, though within permissible skew), we would not be able to conclude that we have become alert. This would not stop prefix selection from doing its job—after all, a header after the window means that we now have a known density—and so we would continue to adopt blocks from the honest chain; however, the malicious node could keep presenting us with a header that is just out of reach, preventing us from ever concluding we are up to date and hence from producing new blocks. The only reason we allow for clock skew at all, however, is to avoid branding nodes as adversarial whereas in fact its just that our clocks are misaligned. This must therefore be a best-effort only: allow for clock skew, but not if this would exceed s slots from the intersection point.

21.5.4 Falling behind

Definition 21.10 gives us a way to discover that we are up to date. Deciding when we are *not* up to date is less critical. One option is to simply use the inverse of definition 21.10 and say we are not up to date when one of our peers provides us with a chain that we cannot see until its tip. Another option is to assume we are not up to date when we boot up the node, and then only conclude that have somehow fallen behind again if we notice that our tip is more than a certain distance away from the wallclock (at most s slots). This may need some special care; if nodes stop producing blocks for a while, we might end up in a state in which we both conclude that we are up to date (because we can see all of our peer's chains to their tip) and not up to date (because our tip is too far from the wallclock). However, this scenario needs special case anyway; we will come back to it in chapter 22.

21.5.5 Block production

In principle, block production can happen even when the node is not up to date. There is however not much point: any blocks that the node will produce while it is not up to date are likely to be discarded almost immediately after production, because the node will prefer the existing (honest) chain over the tiny fork that it itself created. Moreover, blocks that we produce while we are not up to date may in fact be helpful for an adversary. We should therefore disable block production while the node is not up to date.

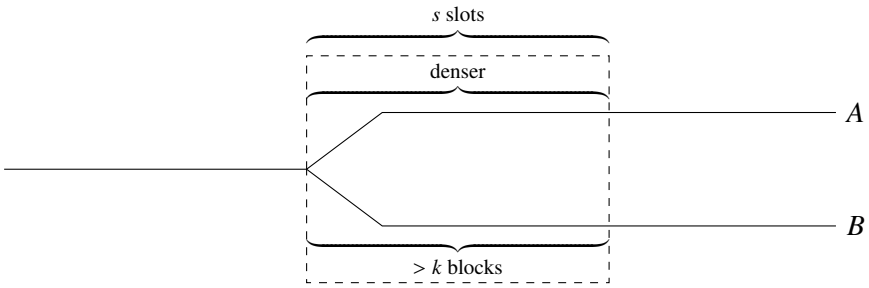
21.6 Implementation concerns

21.6.1 Chain selection in the chain database

We mentioned in section 21.4 that the prefix selection interface works equally well for the chain sync client and the chain database. They are however not doing the same thing; this is a subtle point that deserves to be spelled out in detail.

It comes back to the difference between perceived chain length and actual chain length (section 21.1.2). In the chain sync client this is a meaningful and useful difference: since we are tracking chains from individual peers, it makes sense to distinguish between having seen the tip of that particular chain, or only seeing a prefix of that chain. However, unless we completely change the API to the chain database, the chain database just sees individual blocks, without knowing anything about their provenance; it does therefore not know if those blocks are the tip of "their" chains; it's not even clear what that would mean.

Of course, when the chain database is constructing fragments of chains through its volatile database, it knows if a particular block is the tip of any constructed fragment. However, that is *perceived* chain length: it might be a tip just because we haven't downloaded any more blocks yet. The difference is important. Consider two forks like this:



Chain A is denser in the window, but B nonetheless has more than k blocks in the window (this is entirely possible; a chain of normal density would have $3k$ blocks in the window). The chain sync client knows about both nodes, knows that the chains extend past the window, will wait until it has seen sufficient blocks from both chains (that is, for the first header outside the window), then do a density comparison, find that A is denser, and choose to download the blocks from chain A.

But the chain database cannot replicate any of that reasoning. When blocks from either chain A or chain B are added, as far as the chain database is concerned, those *are* the tips of those chains. This means that is not doing a density comparison, but a chain length comparison. What's worse, if more than k blocks from chain B are added before it sees any blocks from chain A, then it from that point forward be unable to switch to chain A, as this would involve a rollback of more than k blocks.

This is not necessarily problematic: since the chain sync client has more context, it will make the right decision, and only present blocks from chain A to the database. Indeed, as we saw in section 21.5.2, we will in fact download *only* blocks from the honest chain until we are s slots away from the wallclock, at which point we do one final chain selection, and we are up to date. At this point the Density Rule is *anyway* just selecting the longest chain, so the fact that the chain database is effectively doing longest chain selection *always* does not matter.

It does however mean that the block fetch client becomes an important "guardian" of the chain database; they become more tightly coupled than they are currently. This is unfortunate, but not disastrous; it "only" makes the system more difficult to understand. Solving this problem would require rethinking how the chain database works; this is well outside the scope of this chapter.

There is one advantage to this. Section 21.4 describes how the chain database could *in principle* use prefix selection: compute all paths through the volatile database and then use prefix selection to construct a prefix that the node should adopt as its current chain. While this is a useful *specification* of what the chain database must do, in practice we will probably need an equivalent of lemma 11.2 that will allow us to avoid looking at the entire volatile database every time a new block is added to the database. If we however decide that the chain database is just selecting based on chain length *anyway*, then the existing lemma and existing implementation can be used as-is.

21.6.2 Abstract chain selection interface

The current chain selection API compares two chains at a time, and only looks at the tips of those two chains (section 4.1.1). This will obviously have to change; depending on how exactly we want to modify the chain database (section 21.6.1), we must either replace the existing interface with prefix selection as the primitive operation, or else add prefix selection as an additional concept.

One of the reasons we currently only look at the tips of chains is because this simplified treatment of chain selection in the hard fork combinator. This by itself might not be too difficult to change; for example, we could set the `SelectView` of the hard fork block to be an n-ary sum of the `SelectViews` of the various eras. However, it is not a-priori clear what it would mean to apply, say, the Praos rule in one era on the chain, and the Genesis rule in another. This will require some careful thought, though we can probably just sidestep the entire issue and pretend we were using the Genesis rule all along.

21.6.3 Possible optimisations

Chain selection while we are not up to date has some properties that might enable us to implement some performance optimisations. Here we just list some of the possibilities:

- When a node is operational, we try to line up its average-case performance requirements with its worst-case performance requirements, since this avoids an attack vector: if the average-case performance would be significantly better than the worst-case, it is likely that nodes would be optimised for the average case (for instance, run on hardware that can handle the average case, but not necessarily the worst case); then if a malicious node can intentionally cause the worst-case, they might be able to bring down parts of the network.

  For this reason we don't normally share work between various peers; when multiple upstream peers all send us the same header, we verify the header each time. This means that the average case (most upstream chains are the same) and the worst case (every upstream chain is different) are equal.

  However, it is less important that we can predict accurately how long it takes a node (that isn't up to date) to sync with the network. Such a node is anyway not producing blocks; here, faster is simply better. This means that while we we are not up to date we could share the validation work across upstream peers: when two peers send us the same header, we do not need to validate it twice.

  This is *especially* important when we are not up to date, because due to the requirement to have at least N_{rs} upstream peers, we might be connecting to more peers than usual. Moreover, under normal circumstances we expect all of these peers to present us with exactly the same chain (and finally, these cryptographic checks are expensive).

- Similarly, since we expect all upstream nodes to report the same chain, if we receive a bunch of headers from peer 1, we can just ask peer 2 whether they have the most recent of those headers on their chain, thereby skipping over large chunks of the chain altogether.

- Since we only ever fetch blocks strictly in order, we can simplify the interaction with the block fetch client: it might be easier to generate longer fetch ranges, as well as spread the load more evenly across the peers.

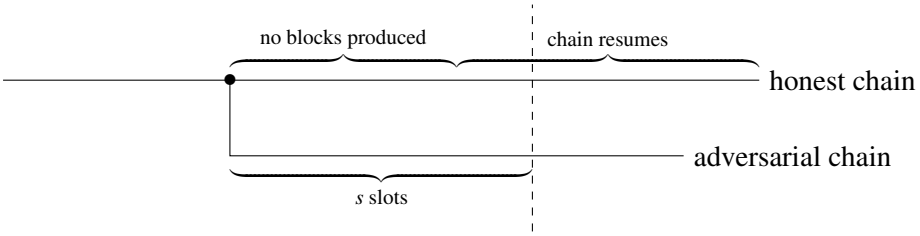
- We saw in section 21.5.2 that any blocks that we download while syncing which are more than s slots away from the wall clock, will be blocks from the common prefix of the honest chains and will not have to be rolled back. It might therefore be possible to bypass the volatile database entirely. However, how this works when we switch back from being up to date to not being up to date would require careful thought.

Chapter 22

Dealing with extreme low-density chains

22.1 Introduction

As we saw in chapter 21, chain density is our principal means for distinguishing chains forged by malicious nodes from the honest chain: due to the fundamental assumption that there is an honest majority, the chain forged by the honest nodes will be denser than any chain forged by an adversary. If therefore the honest nodes in the system stop producing blocks due to some network-wide problem—pervasive node misconfiguration, bug in the ledger, etc.—the security of the system is at risk. Even when the nodes start producing blocks again, the low-density region of the chain left behind by the problem will remain to be an issue for security. If an adversary forks off their own chain at the point where the honest majority stopped producing blocks, then new nodes joining the network (using the genesis rule, chapter 21) will adopt the adversarial chain instead of the honest chain:



The *Disaster Recovery Plan*[56] sketches how we might "patch the chain back up" when a major problem like this occurs. This must happen out-of-band with the cooperation of the major stake holders, and is (mostly) outside the scope of the Consensus Layer report. That said, the options for disaster recovery are currently limited due to a technical limitation in the consensus layer, which we will discuss now.

From a chain security point of view, it makes little difference if the honest chain has a region of s slots containing *one* block or *zero* blocks; both are equally terrible. However, this makes a big difference to the implementation as it currently stands: as long as there is at least one block in every s slots, the system can continue; but when there is a gap of more than s slots anywhere on the chain, the system will grind to a halt. As we will see in this chapter, this is a consequence of the fact that we validate headers independent from blocks, the so-called header/body split (see also section 3.1.1). The main goal of this chapter is to discuss how we can address this, allowing the system to continue irrespective of any gaps on the chain. This is important for a number of reasons:

1. It makes disaster recovery less immediately urgent: if the honest nodes stop producing blocks for whatever reason, the problem can be resolved, the system restarted, and blocks can be produced again. Disaster recovery, and patching the chain back up, can then be considered as the system is running again, and put into motion when the various stake holders are ready.

[56] Currently not available as a public document.

2. It also opens up more avenues for disaster recovery. If the consensus layer can't skip past large gaps on the chain, then the chain *must* be patched. However, if we lift this restriction, then there are other ways in which we might address the problem. For example, we could (even if just temporarily) simply record the low-density area of the chain within the code itself and hardcode a preference for (this part of the) "honest but sparse" chain in chain selection.

3. Chain regions with extreme low density are difficult to avoid in our consensus tests (chapter 19).

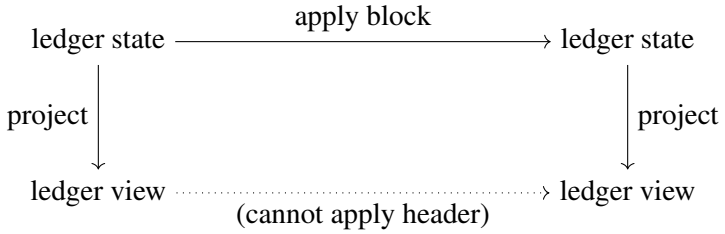
Even *if* it is desirable that the system stops when the chain density falls below a certain threshold, it does not make sense to set that threshold at the "less than 1 block per s slots" boundary. This should be defined and implemented as an external policy, not dictated by implementation details. Moreover, even with an explicit stop, we might like the ability to mark the known-to-be-low-density chain and restart the system (point 2, above). It is also far from clear how to avoid adversarial nodes from taking advantage of such "automatic" stops (how do we prevent adversaries from producing blocks?). Either way, such concerns are well outside the scope of this chapter. Here we address just one question: how can we allow the system to continue when there are larger-than-s-slots gaps on the chain.

22.2 Background

22.2.1 Recap: ledger state, ledger view and forecasting

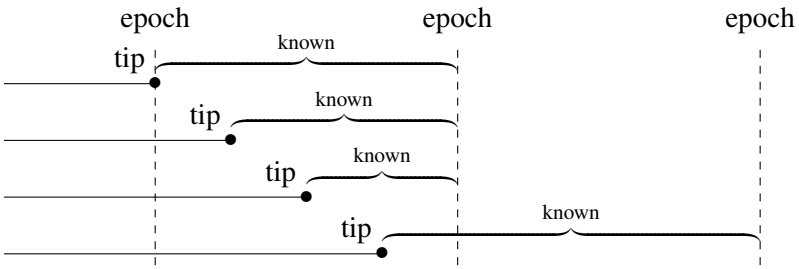
Blocks are validated against the state of the ledger (section 5.1.2). For example, we check that inputs spent by transactions in the block are available in the UTxO in the ledger state. Depending on the choice of consensus protocol, we may also need part of the ledger state to be able to validate the block *header*. For example, in Praos and Genesis we need to know the active stake distribution in order to be able to verify that whoever produced the block had a right to do so. We call the part of the ledger state that we need to validate block headers the *ledger view* (section 4.2.2).

We call it a ledger *view* because it is a projection out of the full ledger state. Unfortunately, we cannot compute the *next* ledger view based only on the header; there is nothing that corresponds to the dotted arrow in this diagram:



Let's recall the Praos example again: we can compute the active stake distribution from the ledger state, but in order to understand how the active stake distribution evolves, we need to know how the full UTxO evolves, and for that we need the full blocks. (We discussed this also in section 18.3.1.)

Let's stay with Praos a little longer. The active stake distribution changes only at epoch boundaries. Therefore we will know the active stake distribution at least until the end of the epoch. Moreover, once we get close enough to the epoch boundary, we also know the stake distribution for the *next* epoch. The range over which we know the active stake distribution therefore evolves as follows:

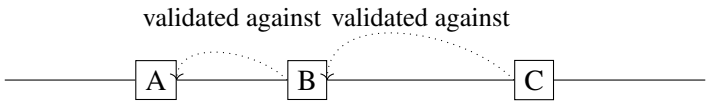


The range over which we know the active stake distribution shrinks and then grows again, but never falls below a certain minimum size. We abstract from this process in the consensus layer, and say we can *forecast* the ledger view from a particular ledger state over a certain *forecast range* (section 5.1.4). This does not necessarily mean the ledger view is constant during that range, but merely that any changes are *known* (for example, see the last line in the diagram above).

If we change our perspective slightly, we can say that blocks on the chain cannot influence the ledger view (active stake distribution) until a certain period of time (in slots) has passed. We call this the *stability window* of the ledger, and will study it in more detail in the next section.

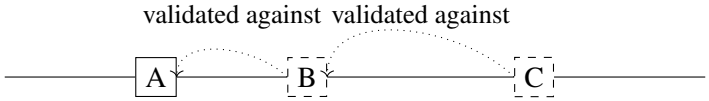
22.2.2 Recap: stability windows

Blocks are validated against ledger states; each block is validated against the ledger state as it was after applying the previous block. This means that when we validate block B in the example below, we use the ledger state after applying block A; for block C, we use the ledger state after applying block B:

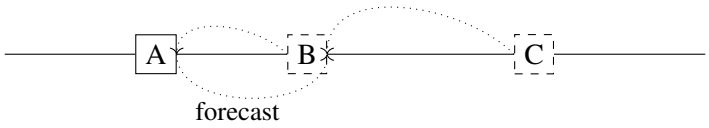


Horizontal axis represents time (in slots)

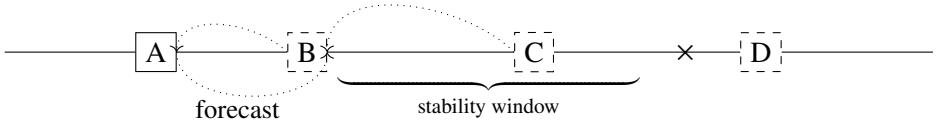
In the chain sync client (chapter 14) we are however not validating blocks, but block *headers*. As we saw, in order to validate a header we only need part of the ledger state, known as the ledger *view*. We also saw that despite the fact that we only need part of the ledger state, we cannot *update* the ledger view using only headers: we still need the full block. This means that if we have block A, but only block *headers* B and C, we have a problem:



Validating header B is unproblematic, since we have the ledger state available after applying block A. However, since we don't have block B, we can't compute the ledger state after block B to validate header C. We are saved by the fact that we can *forecast* the ledger view required to validate header B from the ledger state after A:



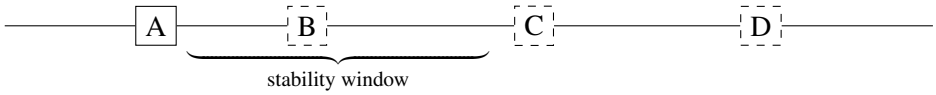
We can do this because of a restriction on the ledger: blocks cannot affect the ledger view until a *stability window* has passed:



We can use the ledger state after applying block A (which we have complete knowledge of) to validate any header up to the end of B's stability window: any changes that A (or any block before A) initiates we know about, and any changes that B initiates cannot take effect until that stability window ends. Therefore we can validate header C, but not header D: block B might have scheduled some changes to take effect at the slot marked as ($\times$) in the diagram, and we do not know what those effects are.[57]

[57] It might be tempting to think that we can validate D because if we did have blocks B and C, block D would be evaluated against the ledger state as it was after applying C, which is still within B's stability window. However, the slot number of D (its location on the x-axis in the diagram) matters, because changes are scheduled for slots.

In chain sync we do not currently take advantage of the knowledge of the location of header B.[58] This means we have to be conservative: all we know is that there could be *some* block in between A and C that might schedule some changes that are relevant for validating header C. In this case we therefore assume that the stability window extends from A instead:



In this example, that means we can validate B, but not C (nor D).[59]

22.2.3 Tension with chain selection

Changes that affect the ledger view are scheduled for slots (often for epoch boundaries, which happen at particular slots); the stability window must therefore be defined in terms of slots as well. This means that the number of *headers* we can validate within a given stability window depends on the density of that chain; if the chain we considered at the end of the previous section looks like this instead



we can validate headers B and C (but still not D).

There is a fundamental tension between the stability window defined in *slots*, and chain selection preferring longer chains: chains that have more *blocks*. In order to be able to do a meaningful comparison between our chain and the candidate chain, we must be able to verify enough of that candidate chain that the length of that verified prefix exceeds the length of our own chain. Since the maximum rollback we support is k (section 4.1.2), that means we must be able to validate at least $k + 1$ headers. The tension is resolved by a theoretical result that says that within $3k/f$ slots we *will* see more than k blocks (more precisely, the probability that we see fewer than k blocks in $3k/f$ slots is negligibly small; [6]). This therefore provides us with a suitable choice for a stability window.

Unfortunately, while in theory there is no difference between theory and practice, there is in practice. Currently, when all nodes in the system are unable to produce blocks for an extended period of time, the system grinds to a halt. Even if the underlying problem is resolved, nodes will refuse to create a block if the distance between that block and the previous block exceeds the stability window; after all, if they did produce a block, other nodes would be unable to validate it. The former is easily resolved, this is merely a check in the block production code; resolving the second problem is the topic of this chapter.

It would be preferable to avoid the tension altogether, and schedule changes that affect the ledger view for particular *blocks* instead (and consequently, have epoch boundaries also happen at certain blocks). This however requires backing from theoretical research; we will come back to this in section 24.4.

[58] We should change this. By anchoring the stability window at the last known block, we only have a guarantee that we can validate k headers, but we should really be able to validate $k + 1$ headers in order to get a chain that is longer than our own (section 22.2.3). If we anchored the stability window after the first unknown header, where it *should* be anchored, we can validate k headers *after* the first unknown header, and hence $k + 1$ in total. Concretely, we would have to extend the `LedgerSupportsProtocol` class with a function that forecasts the ledger view given a *ticked* ledger state. Taking advantage of this would then just be a minor additional complication in the chain sync client.

[59] We could in principle shift this up by 1 slot: after all, the very first next block after A cannot be in the same slot as A. While EBBs are an exception to that rule (chapter 23), we do not need to validate EBBs so this is a rare example where EBBs do not cause a problem.

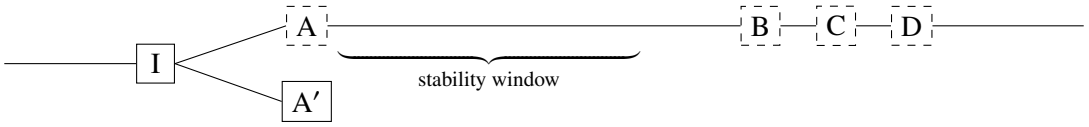
22.2.4 Single-gap case

It is tempting to think that when there is only a *single* large gap on the chain, there is no problem:



The gap between A and B exceeds the stability window, but this should not matter: it's not the stability window after A that matters, but the stability window after B. This seems to be a useful special case: if a problem *does* arise that prevents nodes from producing blocks for an extended period of time, one might hope that this problem does not immediately arise again after the nodes resume producing blocks.

As we saw, the consensus layer always conservatively anchors the stability window at the last known block rather than the first header after the tip. We could change this (and probably should; see footnote 58), but it turns out this does not actually help very much for this particular problem. To see this, suppose there is another node in the system which is currently on a fork that intersects with this chain after some block I before the gap:



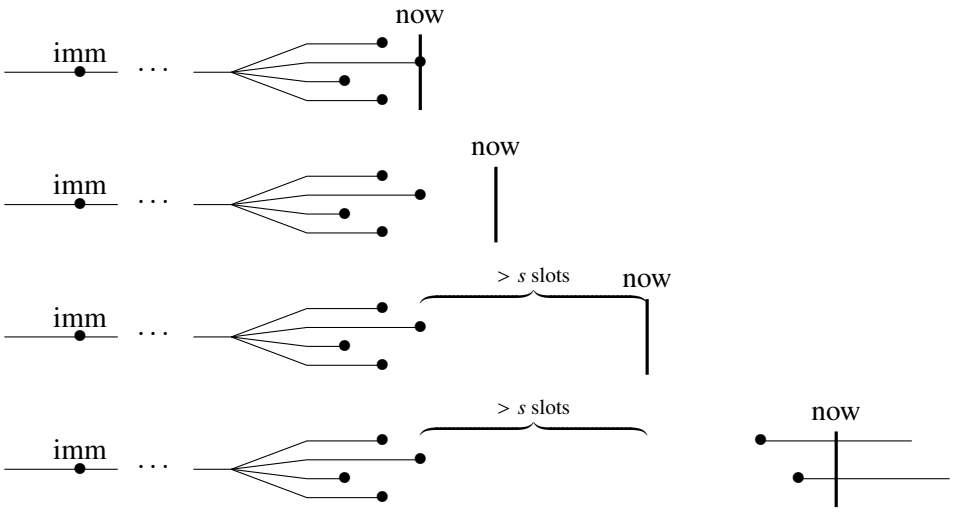
The second node must execute a rollback to I in order to be able to adopt the new chain, but from *its* perspective the first unknown block is A, not B: hence the stability window *must* be anchored at A, and the node will be unable to bridge the gap.

22.3 Pre-genesis

In this section we will consider how we might allow nodes to recover from a low density chain, prior to the implementation of the genesis rule. An obvious solution suggests itself: we could just allow chain sync to download blocks when it needs to validate a header which is beyond its forecast range.

22.3.1 Damage mitigation

The reason the chain sync client doesn't normally download blocks is to limit the amount of unnecessary work an attacker can make it do (prevent DoS attacks, section 3.1.1). We might therefore consider if we can restrict *when* we allow the chain sync client to download blocks. Ideally we would do this only "when necessary": to bridge the gap on the honest chain. Unfortunately, it is difficult to come up with a criterion that approximates this ideal. Consider how the situation evolves from the point of view of a single node:



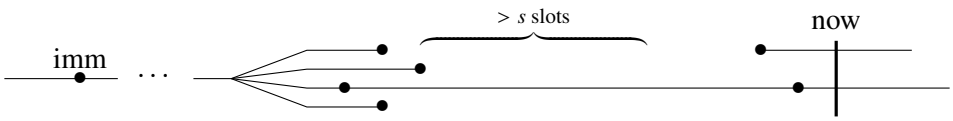
The node is tracking the chains of a number of upstream peers. These chains will share some common prefix, which must at least include the tip of our own immutable database (that is, the block k blocks away from our tip), marked "imm". When block production is halted due to some problem, the gap between the tips of the chains and the wallclock will start to increase; at some point this gap will exceed the stability window. Finally, when the problem is resolved the nodes will start producing blocks again.

Assumption 22.1. In the period where the honest nodes cannot produce any blocks, malicious nodes cannot either. If that is not the case, we are in trouble anyway; that is a problem which is well outside the scope of this chapter.

Assumption 22.1 seems to give some hope. We may not be able to decide for any *particular* chain if that chain happens to be the honest chain. However, if *none* of the chains contain any blocks in the gap, then eventually it will be true for *all* upstream peers that the gap from the tip of that peer's chain to the wallclock exceeds the stability window. This might suggest the following rule:

Failed attempt 22.1. Only allow the chain sync client to download blocks if this would be required for *all* peers.

Unfortunately, this rule does not work because as soon as we bridge the gap for *one* of our peers, that condition no longer holds:



Now one of our chains has a tip which is near the wallclock, and so the condition no longer holds. Okay, you might say, but it was true at *some* point, and when it was true, it would have allowed the chain sync client to download blocks for *any* peer. Thus, we could try the following rule:

Failed attempt 22.2. When we detect that the tips of all upstream peers are more than the stability window away from the wallclock, give the chain sync client a chance to download blocks for *all* peers.

This *might* work, but it's very stateful. What does "all peers" mean exactly? All peers we are currently connected to? What if we connect to another peer later? What if the node has restarted in the meantime, do we need to persist this state? Will we need some notion of peer identity? Perhaps all of these questions have answers, but this does not seem like a clean solution.

As a final attempt, we might try to ensure that there is only a *single* chain after we resolve the problem that was preventing block production. Suppose this could somehow be guaranteed (out of band communication to agree on a block in the common prefix, use a BFT-like leadership selection for a while, etc.). Then we could try the following rule:

Failed attempt 22.3. When we detect that the tips of all upstream peers are more than the stability window away from the wallclock, allow the chain sync client to download enough blocks to bridge the gap for *one* peer. Allow the other peers to bridge the gap only if they contain the *same* header after the gap.

Unfortunately, this still cannot work. Even if the honest nodes agree to only produce a single chain after the gap, we cannot prevent an adversary from constructing another chain. If the node then happens to pick the adversary's chain as the one-and-only allowed header to jump the gap, it would be unable to then switch to the honest chain later.

22.3.2 Damage analysis

If we cannot limit when the chain sync client is allowed to download and validate blocks, then let's analyse exactly what the possibility for denial of service attacks really is.

Lemma 22.1. When the node is up to date, the chain sync client will never have to download any blocks.

Proof. The Praos analysis [6] tells us that the honest chains will not diverge by more than k blocks, and that this means that their intersection cannot be more than $3k/f$ slots away from the wallclock (provided block production is not halted, of course). This means that any header that would be more than the stability window away from the intersection point would have a slot number past the wallclock, and would therefore be invalid.[60] □

This means that we only have to worry about DoS attacks while a node is syncing. As a first observation, node performance is less critical here. The node is anyway not producing blocks while syncing, so causing the node to slow down temporarily is not a huge deal (*cf.* also section 21.6.3 where we argue it's less essential during syncing to make the worst case performance and the normal case performance the same).

It will therefore suffice to simply *bound* the amount of work a malicious node can make us do. We have to make sure that we can see at least $k + 1$ headers from each peer (we want to support a rollback of k blocks, and chain selection is based on length, so if we can validate $k + 1$ headers, we have seen enough to do a length comparison and decide we want to switch to the other chain). This means we would need to download at most k blocks.

This bounds the amount of *memory* we might need to dedicate to any chain,[61] but does not limit how much *work* they can make us do: an attacker with even a small amount of stake could construct lots of chains that fork off the main chain, and so we'd end up downloading and validating lots of blocks. We can limit the impact of this by rate limiting rollback messages, which would be useful for other purposes as well.[62] Moreover, there is no real asymmetry here between the attacker and the defender: the cost of downloading and validating a block on our side is not too dissimilar from the cost of producing and providing that block on the side of the attacker, and all the attacker would gain in doing so is slow down a node's syncing speed. (Admittedly, if we adopt more than k blocks from the adversarial chain we'd be in trouble, but that is a problem solved by the Genesis chain selection rule).

[60] Though we allow for some minimal clock skew, headers past the wallclock should be considered invalid if this exceeds s slots from the immutable tip, even if they would still fall within the permissible clock skew. This is an edge case that was important for implementation of genesis as well; see section 21.5.3.

[61] Currently the length of the fragments we keep in memory for each upstream peer is bound by the forecast range, but that natural bound would of course no longer work if we allow the chain sync client to download blocks.

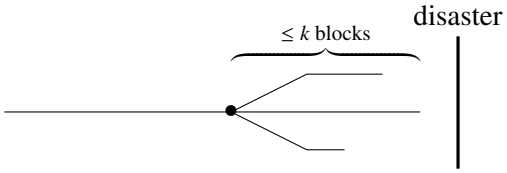
[62] For example, it can help avoid a DoS attack where an attacker attempts to flood our volatile DB with lots of useless blocks.

22.4 Post-genesis

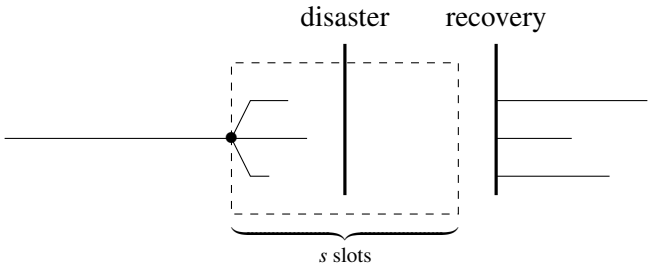
With the implementation of the genesis rule, discussed in detail in chapter 21, some things get easier, but unfortunately some things get more difficult.

22.4.1 Pre-disaster genesis window

Suppose the chain is operating as normal until disaster strikes and the nodes stop producing blocks:



While the Genesis analysis [1] tells us that that common intersection point is *at most* k blocks away, in practice it will actually be much less than k most of the time, a handful of blocks in typical cases. This means that when the nodes start producing blocks again, chain selection will be a looking at a window of s slots where all chains have very low density:[63]



In effect we are doing a density comparison over very short fragments. In general this is not meaningful; in the extreme case, where that fragment contains only a single slot, density will either be 100% or 0%. It is tempting to think that we could just *grow* the genesis window to include part of the post-disaster chain. Growing the genesis window is however not sound: once we get more than s slots away from the intersection point, an adversary can start to influence the leadership schedule and so density comparisons are no longer meaningful.

Essentially what this means is that after disaster recovery we arbitrarily pick any of the chains from before the disaster to continue. This probably does not matter too much; at worst more blocks are lost than strictly necessary, but those transactions can be resubmitted and we're anyway talking about disaster recovery; some loss is acceptable. Verify

It might *even* okay if the chain we happened to pick was constructed by an adversarial node. After all, at most they can have constructed k blocks, and all they can do is selectively *omit* transactions; if we continue the chain based on such an adversarial chain, the damage they can do is very limited. Verify

However. Suppose we do make an arbitrary choice and the chain resumes. Nothing is preventing an adversary from forking off a new chain just prior to the disaster region *after the fact*. If they do, and new nodes joining the system end up choosing that chain, they are in serious trouble; now they are following a chain that is basically under the control of the adversary.

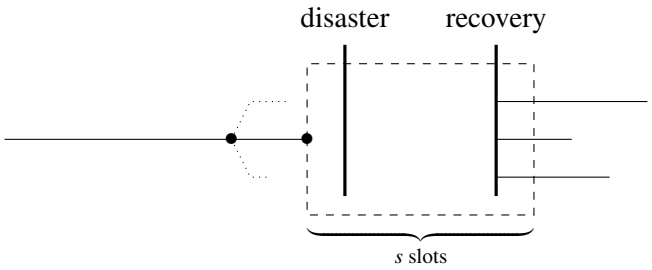
[63] Prefix selection does a length comparison when we can see all chains to their tip, meaning all chains terminate within the s window. It is important that we don't reinterpret that as "all chains are less than k *blocks* away from the intersection point". If we did, we would conclude in this case that we can still do a length comparison when the chains continue after the end of the disaster period; that is not correct: it would mean that while the chains start are growing we would come to one conclusion, but then once the chains grow past the window of k blocks, we would switch to comparing density and might come to a *different* conclusion.

This ability of adversaries to construct new forks before areas of low density on the chain mean that these areas are a serious risk to security. Indeed, somewhat ironically this risk is made *worse* by the genesis rule. If we look at chain length only, the honest chain will probably be longer than whatever chain an attacker forges; but if we look at density, an attacker than can even produce a single block in s slots might already have a sufficient advantage.

This means that some kind of disaster recovery becomes even more important after we implement the genesis rule. Ideally we would patch the chain up, but there is an easier option which can work (at least as a temporarily solution): it suffices to hardcode a pre-disaster block as the agreed-on pre-disaster tip.

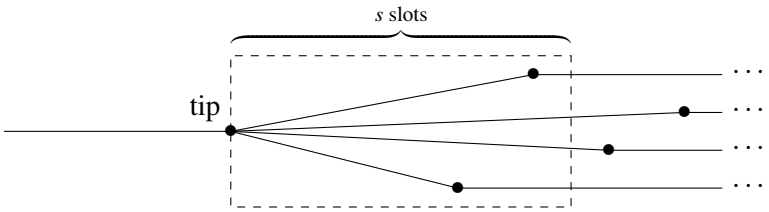
22.4.2 Post-disaster genesis window

So far we've been talking about the genesis window as we approach the disaster. Suppose we choose *some* block as our pre-disaster tip; either by randomly selecting one of the chains (or if by luck all chains happen to converge pre-disaster) or by hardcoding a preference for a certain block:

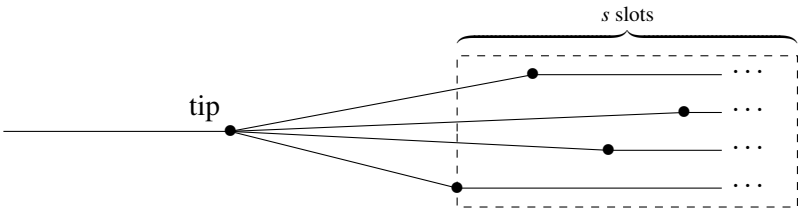


Having made this choice, we are *again* faced with a comparison between chains which all have very low density within the window (in the extreme case, even zero). This means that here we effectively have a *second* arbitrary choice between chains, with all the same dangers (in particular the danger of an attacker forking off a new chain after the fact). However, in this case we have a way out:

Lemma 22.2. Suppose we have decided on a particular pre-disaster tip, and the chains we see look like this:



Then we can shift up the genesis lookahead window until it starts at the first block after the tip:



Proof. The first block that could be produced by an adversary is the first block after the tip. This adversarial block cannot influence the leadership schedule until at least $3k/f$ slots later, which is also the size of the lookahead window (s). Therefore a density comparison within the shifted window will still favour the honest chains. □

Lemma 22.2 means that we can shift the genesis window until after the disaster, and avoid the second arbitrary choice between chains. In particular, it means we can definitely make it across the gap safely if we *mark* the before-disaster block (to avoid picking an adversary's block).

22.4.3 (No) need for gap jumping

In section 22.3 we discuss that prior to the implementation of the genesis rule, we sometimes need to allow the chain sync client to download blocks. Since chain selection was based on length, we need to be able to validate a sufficient number of headers to get a fragment that is longer than our current chain; in the case of a disaster, that might mean validating a header that is more than s slots away from our latest usable ledger state to validate that header, and hence we may need to download some blocks.

The genesis rule, in principle, *never needs to look past s slots*. It makes all of its decisions based on a window of s slots; if a node reports a header past the end of that window, that just tells us we have seen everything we need to see about that chain within the window. There is no need to validate this header: any headers *within* the window contribute to the density of the chain and are validated, any headers *past* the window just cap that density; nodes cannot increase their chain's density with an invalid header past the window, and so nothing can go wrong if we do not validate that header.

This changes however if we want to make use of lemma 22.2. It is of course necessary that we validate the headers *within* the window; if we shift the window, we are no longer guaranteed that "within the window" is synonymous with "at most s slots away from the ledger state we have available".

Whether or not this opens us up to denial of service attacks depends on when exactly we shift the window. However, if we do this only if we have some kind of explicit disaster recovery (where we mark the pre-disaster block), or if the density in the window we see drops below a certain threshold, then the scope for a denial is service attack is very limited indeed.

22.4.4 In the absence of adversaries

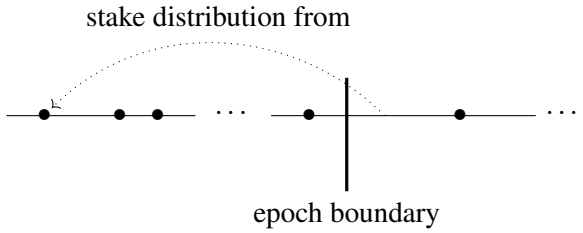
In the consensus tests (chapter 19) periods where no blocks are being produced are hard to avoid. However, we do not (currently) model adversarial behaviour. This means that any kind of explicit disaster recovery is not needed: if pre-disaster and post-disaster we end up picking an "arbitrary" chain, consensus is still guaranteed. After all, the choice is not "arbitrary" in the sense that different nodes may pick different chains; it is only "arbitrary" in the sense that we are doing a density comparison on a fragment that is too short (it may be necessary to add a deterministic tie-breaker in case there are multiple fragments with equal density).

Chapter 23

Epoch Boundary Blocks

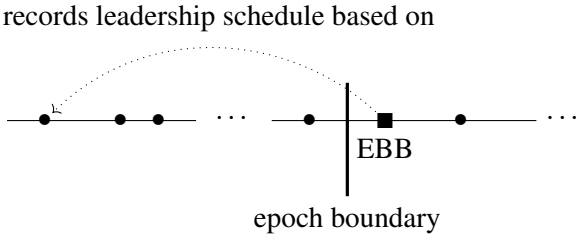
23.1 Introduction

Recall that when a new epoch begins, the active stake distribution in the new epoch—that is, the stake distribution used to determine the leader schedule— is not the stake distribution as it was at the end of the last epoch, but rather as it was some distance back:



This means that blocks cannot influence the active stake distribution until some time in the future. That is important, because when a malicious node forks off from the honest chain, the leadership schedule near the intersection point cannot be influenced by the attacker, allowing us to compare chain density and choose the honest chain (which will be denser because of the assumed honest majority); see chapter 21 for an in-depth discussion.

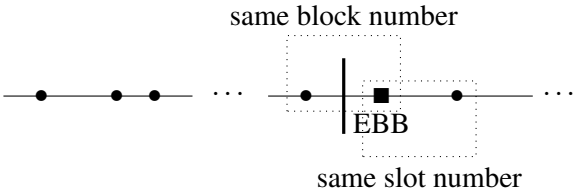
In the literature, the term "epoch boundary block" (or EBB for short) normally simply refers to the last block in any given epoch (for example, see [2]). It might therefore be a bit surprising to find the term in this report since the final block in an epoch is not of special interest in the Ouroboros family of consensus protocols. However, in the first implementation of the Byron ledger (using the original Ouroboros protocol [10], which we now refer to as "Ouroboros Classic"), a decision was made to include the leadership schedule for each new epoch as an explicit block on the blockchain; the term EBB was used to refer to this special kind of block:[64]



[64] It is not entirely clear if an EBB should be regarded as the final block in an epoch, or as the first block in the next epoch. The name would suggest that the former interpretation is more appropriate; as it turns out, however, the very first epoch on the chain *starts* with an EBB, recording the leadership schedule derived from the genesis block. We will therefore regard the EBB as starting an epoch, rather than ending one.

Having the leadership schedule explicitly recorded on-chain turns out not to be particularly useful, however, and the code was modified not to produce EBBs anymore even before we switched from Byron to Shelley (as part of the OBFT hard fork, see section 2.3); these days, the contents of the existing EBBs on the chain are entirely ignored. Unfortunately, we cannot forget about EBBs altogether because—since they are an actual block on the blockchain—they affect the chain of block hashes: the first "real" block in each epoch points to the EBB as its predecessor, which then in turns points to the final block in the previous epoch.

So far, none of this is particularly problematic to the consensus layer. Having multiple types of blocks in a ledger presents some challenges for serialisation (section 6.1.1), but does not otherwise affect consensus much: after all, blocks are interpreted by the ledger layer, not by the consensus layer. Unfortunately, however, the design of the Byron EBBs has odd quirk: an EBB has the same block number as its *predecessor*, and the same slot number as its *successor*:



This turns out to be a huge headache. When we started the rewrite, I think we underestimated quite how many parts of the system would be affected by the possibility of having multiple blocks with the same block number and multiple blocks with the same slot number on a single chain. Some examples include:

- For chain selection protocols based on chain length, two chains may end in blocks with the same block number, yet not have equal length.
- When we validate block headers that contain explicit block numbers, we cannot insist that those block numbers are monotonically increasing, instead having to add a special case for EBBs.
- TODO: Many, many others

In hindsight, we should have tried harder to eliminate EBBs from the get-go. In this chapter, we will discuss two options for modifying the existing design to reduce the impact of EBBs (section 23.2), or indeed eliminate them altogether (section 23.3).

23.2 Logical slot/block numbers

23.3 Eliminating EBBs altogether

Chapter 24

Miscellaneous

TODO: This is a mess at the moment.

TODO

24.1 On abstraction

ledger integration: as things were changing a lot, it made sense for consensus to define the ledger API internally and have the integration be done consensus side. but as things are stabilising, it might make more sense for that abstraction to live externally, so that you can literally plug in Shelley into consensus and we don't have to do anything

24.2 On-disk ledger state

Sketch out what we think it could look like Consequences for the design

Duncan suitable section.

24.3 Transaction TTL

Describe that the mempool could have explicit support for TTL, but that right now we don't (and why this is OK: the ledger anyway checks tx TTL). We should discuss why this is not an attack vector (transactions will either be included in the blockchain or else will be chucked out because some of their inputs will have been used).

24.4 Block based versus slot based

24.5 Eliminating safe zones

Are they really needed? Consensus doesn't really look ahead anymore? (Headers are not checked for time; leadership is ticking, not forecasting). Does the wallet really need it? What about the ledger?

Other thought: what if we split slots into "microslots", 20 microslots to a slot. Now the slot/time mapping is *always* known, and for Shelley etc we don't actually need to know the global microslot, all we care about is the microslot within a slot (and hence is independent of when Shelley starts). This would make time conversion no longer state dependent.

24.6 Eliminating forecasting

This is a stronger version of section 24.5, where we eliminate *all* forecasting. Specifically, this means that we don't do header validation anymore, relying on the chain DB to do block validation. This would be

an important simplification of the consensus layer, but we'd need to analyse what the "benefit" of this simplification is for an attacker. Personally, I think it'll be okay.

The most important analysis we need to do here is how this affects the memory usage of the chain sync client. Note that we already skip the ahead-of-time check, which we don't do until we have the full block and validate it. We should discuss that somewhere as well.

24.7 Open kinds

Avoid type errors such as trying to apply a ledger to a block instead of an era (or an era instead of crypto, or..).

24.8 Relax requirements on time conversions

Perhaps it would be okay if time conversions we strictly relative to a ledger state, rather than "absolute" (section 17.4).

24.9 Configuration

What a mess.

24.10 Specialised chain selection data structure

In section 11.2 we describe how chain selection is implemented. However, in an ideal world this would mean we have some kind of specialised data structure supporting

- Efficient insertion of new blocks
- Efficient computation of the best chain

It's however not at all clear what such a data structure would look like if we don't want to hard-code the specific chain selection rule.

24.11 Dealing with clock changes

When the user changes their system clock, blocks that we previously adopted into our current chain might now be ahead of the system clock (section 11.5) and should not be part of the chain anymore, and vice versa.

When the system clock of a node is moved *forward*, we should run chain selection again because some blocks that we stored because they were in the future may now become valid. Since this could be any number of blocks, on any fork, probably easiest to just do a full chain selection cycle (starting from the tip of the immutable database).

When the clock is moved *backwards*, we may have accepted blocks that we should not have. Put another way, an attacker might have taken advantage of the fact that the clock was wrong to get the node to accept blocks in the future. In this case we therefore really should rollback— but this is a weird kind of rollback, one that might result in a strictly smaller current chain. We can only do this by re-initialising the chain DB from scratch (the ledger DB does not support such rollback directly). Worse still, we have have decided that some blocks were immutable which really weren't.

Unlike the data corruption case, here we should really endeavour to get to a state in which it was as if the clock was never "wrong" in the first place; this may mean we might have to move some blocks back

from the immutable DB to the volatile DB, depending on exactly how far the clock was moved back and how big the overlap between the immutable DB and volatile DB is.

It is therefore good to keep in mind that the overlap between the immutable DB and volatile DB does make it a bit easier to deal with relatively small clock changes; it may be worth ensuring that, say, the overlap is at least a few days so that we can deal with people turning back their clock a day or two without having to truncate the immutable database. Indeed, in a first implementation, this may be the *only* thing we support, though we will eventually have to lift that restriction.

Right now, we do nothing special when the clock moves forward (we will discover discover the now valid blocks on the next call to `addBlock` (section 11.4). When the clock is reset *backwards*, the node will currently (intentionally) crash, we make no attempt to try and reset the state (the current slot number moving backwards might cause difficulties in many places). Unfortunately, if the clock is moved so far back that blocks in the *immutable database* are now considered to be ahead of the wall clock, we will not currently detect this (section 17.6.3).

Part VII

Conclusions

Chapter 25

Technical design decisions

In this chapter we will discuss a number of interesting technical decision decisions that aren't directly to any of the specific needs of the consensus layer.

25.1 Classes versus records

Discuss why classes are helpful (explicit about closures).

25.2 Top-level versus associated type families

Chapter 26

Conclusions

Part VIII

Appendices

Appendix A

Byron

Some details specific to the Byron ledger. EBBs already discussed at length in chapter 23.

The Byron specification can be found at `https://github.com/IntersectMBO/cardano-ledger`.

A.1 Update proposals

A.1.1 Moment of hard fork

The Byron ledger state provides the current protocol version in

```
adoptedProtocolVersion :: ProtocolVersion
```

in the `State` type from `Cardano.Chain.Update.Validation.Interface`. This protocol version is a three-tuple *major*, *minor*, *alt*. The Byron specification does not provide any semantic interpretation of these components. By convention (outside of the purview of the Byron specification), the hard fork is initiated the moment that the *major* component of `adoptedProtocolVersion` reaches a predefined, hardcoded, value.

A.1.2 The update mechanism for the `ProtocolVersion`

Updates to the `ProtocolVersion` in Byron are part of the general infrastructure for changing protocol parameters (parameters such as the maximum block size), except that in the case of a hard fork, we care only about changing the `ProtocolVersion`, and not any of the parameters themselves.

The general mechanism for updating protocol parameters in Byron is as follows:

1. A protocol update *proposal* transaction is created. It proposes new values for some protocol parameters and a greater *protocol version* number as an identifier. There cannot be two proposals with the same version number.

2. Genesis key delegates can add *vote* transactions that refer to such a proposal (by its hash). They don't have to wait; a node could add a proposal and a vote for it to its mempool simultaneously. There are only positive votes, and a proposal has a time-to-live (see `ppUpdateProposalTTL`) during which to gather sufficient votes. While gathering votes, a proposal is called *active*.

   Note that neither Byron nor Shelley support full centralisation (everybody can vote); this is what the Voltaire ledger is intended to accomplish.

3. Once the number of voters satisfies a threshold (currently determined by the `srMinThd` field of the `ppSoftforkRule` protocol parameter), the proposal becomes *confirmed*.

4. Once the threshold-satisfying vote becomes stable (i.e. its containing block is at least $2k$ slots deep), a block whose header's protocol version number (`CC.Block.headerProtocolVersion`) is that of the proposal is interpreted as an *endorsement* of the stably-confirmed proposal by the block's issuer

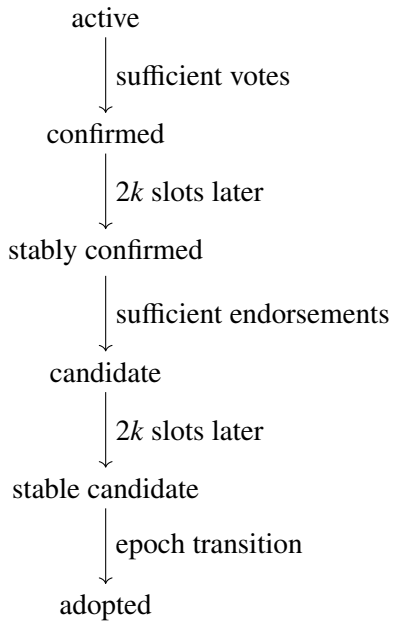


Figure A.1: Byron update proposal process

(specifically by the Verification Key of its delegation certificate). Endorsements—i.e. *any block*, since they all contain that header field—also trigger the system to discard proposals that were not confirmed within their TTL.

Notably, endorsements for proposals that are not yet stably-confirmed (or do not even exist) are not invalid but rather silently ignored. In other words, no validation applies to the ‘headerProtocolVersion‘ field.

5. Once the number of endorsers satisfies a threshold (same as for voting), the confirmed proposal becomes a *candidate* proposal.

6. *At the beginning of an epoch*, the candidate proposal with the greatest protocol version number among those candidates whose threshold-satisfying endorsement is stable (i.e. the block is at least $2k$ deep) is *adopted*: the new protocol parameter values have now been changed.

   If there was no stable candidate proposal, then nothing happens. Everything is retained; in particular, a candidate proposal whose threshold-satisfying endorsement was not yet stable will be adopted at the subsequent epoch unless it is surpassed in the meantime.

   When a candidate is adopted, all record of other proposals/votes/endorsements—regardless of their state—is discarded. The explanation for this is that such proposals would now be interpreted as an update to the newly adopted parameter values, whereas they were validated as an update to the previously adopted parameter values.

The diagram shown in fig. A.1 summarises the progress of a proposal that’s eventually adopted. For other proposals, the path short circuits to a “rejected/discarded” status at some point.

A.1.3 Initiating the hard fork

Proposals to initiate the hard fork can be submitted and voted on before all core nodes are ready. After all, once a proposal is stably confirmed, it will effectively remain so indefinitely until nodes endorse it (or it gets superseded by another proposal). This means that nodes can vote to initiate the hard fork, *then* wait for everybody to update their software, and once updated, the proposal is endorsed and eventually the hard fork is initiated.

Endorsement is somewhat implicit. The node operator does not submit an explicit "endorsement transaction", but instead restarts the node[65] (probably after a software update that makes the node ready to support the hard fork) with a new protocol version (as part of a configuration file or command line parameter), which then gets included in the blocks that the node produces (this value is the `byronProtocolVersion` field in the static `ByronConfig`).

A.1.4 Software versions

The Byron ledger additionally also records the latest version of the software on the chain, in order to facilitate software discovering new versions and subsequently updating themselves. This would normally precede all of the above, but as far as the consensus layer is concerned, this is entirely orthogonal. It does not in any way interact with either the decision to hard fork nor the moment of the hard fork. If we did forego it, the discussion above would still be entirely correct. As of Shelley, software discovery is done off-chain.

The Byron *block header* also records a software version (`headerSoftwareVersion`). This is a legacy concern only, and is present in but ignored by the current Byron implementation, and entirely absent from the Byron specification.

[65] A node restart is necessary for *any* change to a protocol parameter, even though most parameters do not require any change to the software at all.

Appendix B

Shelley

B.1 Update proposals

B.1.1 Moment of the hard fork

Similar to the Byron ledger (section A.1.1), the Shelley ledger provides a "current protocol version", but it is a two-tuple (not a three-tuple), containing only a *hard fork* component and *soft fork* component:

```
_protocolVersion :: (Natural, Natural)
```

in `PParams`. The hard fork from Shelley to its successor will be initiated once the hard fork component of this version gets incremented.

B.1.2 The update mechanism for the protocol version

The update mechanism in Shelley is simpler than it is in Byron. There is no distinction between votes and proposals: to "vote" for a proposal one merely submits the exact same proposal. There is also no separate endorsement step (though see section B.1.3).

The procedure is as follows:

1. As in Byron, a proposal is a partial map from parameters to their values.
2. During each epoch, a genesis key can submit (via its delegates) zero, one, or many proposals; each submission overrides the previous one.
3. "Voting" (submitting of proposals) ends $6k/f$ slots before the end of the epoch (i.e., twice the stability period, called `stabilityWindow` in the Shelley ledger implementation).
4. At the end of an epoch, if the majority of nodes (as determined by the `Quorum` specification constant, which must be greater than half the nodes) have most recently submitted the same exact proposal, then it is adopted.
5. The next epoch is always started with a clean slate, proposals from the previous epoch that didn't make it are discarded.[66]

The protocol version itself is also considered to be merely another parameter, and parameters can change without changing the protocol version, although a convention could be established that the protocol version must change if any of the parameters do; but the specification itself does not mandate this.

[66] Proposals *can* be explicitly marked to be for future epochs; in that case, these are simply not considered until that epoch is reached.

B.1.3 Initiating the hard fork

The timing of the hard fork in Shelley is different to the one in Byron: in Byron, we *first* vote and then wait for people to get ready (section A.1.3); in Shelley it is the other way around.

Core node operators will want to know that a significant majority of the core nodes is ready (supports the hard fork) before initiating it. To make this visible, Shelley blocks contain a protocol version. This is not related to the current protocol version as reported by the ledger state (`_protocolVersion` as discussed in the previous section), but it is the *maximum* protocol version that the node which produced that block can support.

Once we see blocks from all or nearly all core nodes with the 'hard fork' component of their protocol version equal to the post-hard-fork value, nodes will submit their proposals with the required major version change to initiate the hard fork.[67]

B.2 Forecasting

Discuss the fact that the effective maximum rollback in Shelley is $k - 1$, not k; see also section 5.2.

[67] This also means that unlike in Byron (footnote 65), in Shelley there is no need to restart the node merely to support a particular parameter change (such as a maximum block size).

Bibliography

[1] Badertscher, C., Gazi, P., Kiayias, A., Russell, A., and Zikas, V. Ouroboros Genesis: Composable proof-of-stake blockchains with dynamic availability. Cryptology ePrint Archive, Report 2018/378, 2018. https://eprint.iacr.org/2018/378.

[2] Buterin, V., Hernandez, D., Kamphefner, T., Pham, K., Qiao, Z., Ryan, D., Sin, J., Wang, Y., and Zhang, Y. X. Combining GHOST and Casper, 2020.

[3] Castro, M., and Liskov, B. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst. 20*, 4 (Nov. 2002), 398–461.

[4] Coutts, D., David, N., Szamotulski, M., and Thompson, P. Introduction to the design of the data diffusion and networking for Cardano Shelley. Tech. rep., IOHK, August 2020. Version 1.9.

[5] Coutts, D., and de Vries, E. Formal specification for a Cardano wallet. Tech. rep., IOHK, July 2018. Version 1.2.

[6] David, B., Gaži, P., Kiayias, A., and Russell, A. Ouroboros Praos: An adaptively-secure, semi-synchronous proof-of-stake protocol. Cryptology ePrint Archive, Report 2017/573, 2017. https://eprint.iacr.org/2017/573.

[7] Dimjašević, M., and Clark, N. Specification of the blockchain layer. Tech. rep., IOHK, May 2019. Part of the Byron specification, available from https://github.com/IntersectMBO/cardano-ledger/.

[8] Hinze, R., and Paterson, R. Finger trees: A simple general-purpose data structure. *J. Funct. Program. 16*, 2 (Mar. 2006), 197–217.

[9] Kiayias, A., and Russell, A. Ouroboros-BFT: A simple Byzantine fault tolerant consensus protocol. Cryptology ePrint Archive, Report 2018/1049, 2018. https://eprint.iacr.org/2018/1049.

[10] Kiayias, A., Russell, A., David, B., and Oliynykov, R. Ouroboros: A provably secure proof-of-stake blockchain protocol. Cryptology ePrint Archive, Report 2016/889, 2016. https://eprint.iacr.org/2016/889.