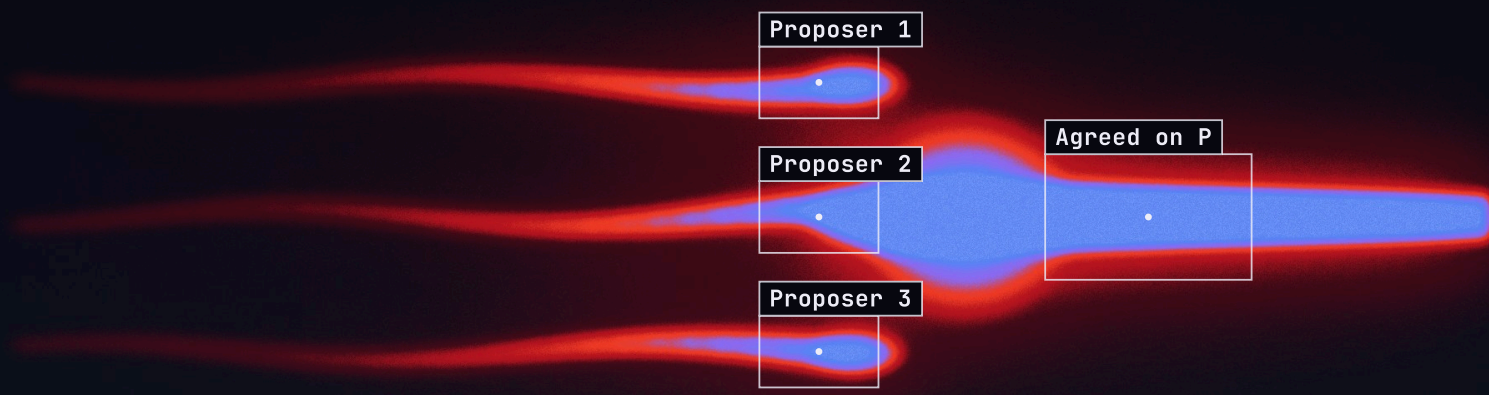
A SOLANA FORK · ITS OWN CHAIN
K 3 · $\delta_{block}$ 50 MS · $2\delta_{local}$ 20 MS
NEW CONSTELLATION CONSENSUS · FRA NYC TYO




# Acrux

## Solana at 50 ms.

### No one gets the last look.

A faster Solana: a fork that runs Solana programs unchanged, with New Constellation as its consensus, three proposers in every 50 ms slot and ACRUX as its native gas and stake.

Whitepaper v1.0

2026-11-16

Abstract

# Solana at 50 ms, and no one gets the last look.

Acrux is a faster Solana. It is its own chain, a fork that runs Solana programs and transactions unchanged on the Solana Virtual Machine, with New Constellation as its consensus: the multiple-concurrent-proposer design Quentin Kniep and Roger Wattenhofer published for the Solana Foundation on 2026-11-14 [1]. A slot lasts 50 ms, about four times shorter than the roughly 215 ms Solana mainnet's slots measure since Alpenglow [13], and it has no leader. In every slot, k = 3 co-located proposers each build a block, stream it out encrypted, exchange signed commitments within $2\delta_{\text{local}}$, and only then release their keys. Validators agree on the set of blocks, decrypt them, merge every transaction into one deterministic order by priority fee, execute, and vote through Alpenglow on consensus blocks of four slots. ACRUX is the chain's native token, its gas and its stake.

This paper describes the chain whole: the protocol as the paper specifies it and as Acrux runs it, with the theorem and lemmas that make it safe; `acrux-node`, the Rust node that implements the paper's four algorithms with Rotor, Votor on consensus blocks, threshold transactions and SVM execution; Acrux Net, the chain itself, live on three hubs in Frankfurt, New York and Tokyo with real geographic delays on the wire, measured over 600 seconds with every one of 2,640 consensus blocks finalized and no skip; the Fault Lab, where six attacks the design defends against end in the outcome the paper proves; the Hub Registry, which runs on Acrux, where it drives the proposer schedule, and on Solana mainnet, where Solana validators register and join hubs with the identity they already hold; the SDK, whose one call sends a transaction no single proposer can read; ACRUX; and the road ahead.

Where the paper leaves a choice open, this paper states what Acrux chose: the wire formats, the cipher and its commitment, the codes, the threshold scheme, the fee split, Rotor's relays for k proposers, the regional ordering of windows, and the base timeout.

# Contents


1 **Solana at 50 ms** 8

1.1 A faster Solana 9
1.2 One author per slot 9
1.3 What the position is worth 10
1.4 The motivation New Constellation states 12
1.5 Designs that answer the same problem 12
1.6 Why now 13
1.7 The insight 14
1.8 New Constellation in one slot 15
1.9 What Acrux is 16
1.10 The rest of this document 19

2 **Model and assumptions** 20

2.1 Validators and stake 21
2.2 The adversary 21
2.3 Network and time 22
2.4 The timing assumption 23
2.5 Cryptographic assumptions 25
2.6 Slots, windows and consensus slots 26
2.7 Hubs, proposers and co-proposers 27
2.8 Alpenglow as New Constellation uses it 28
2.9 Notation 29

3 **Proposing a slot** 32

3.1 Blocks and messages 33
3.2 The proposer, Algorithm 1 35
3.3 Dissemination through Rotor 37
3.4 The hash exchange 38
3.5 Fast handover to the next window 41
3.6 A crashed proposer 42

4 **Validating and voting** 43

4.1 What a validator does with a slot 44
4.2 The validator, Algorithm 2 44
4.3 Validity of a proposer block 46
4.4 Agreement 47
4.5 Executing a slot 48
4.6 Voting on a cblock 49
4.7 Timeouts 51

4.8 Consensus on cblocks 52
4.9 Skips 54
4.10 A window with a crashed proposer 54

5 Ordering, threshold transactions and fees 56
5.1 The merge 57
5.2 Slot capacity and Algorithm 4 58
5.3 Threshold transactions 62
5.4 Fees 65

6 Hubs and the proposer schedule 69
6.1 Why proposers are co-located 70
6.2 Proposer stake 71
6.3 Active hubs 72
6.4 From hubs to windows 74
6.5 Acrux Net's hubs 77

7 Correctness 79
7.1 The model 80
7.2 Correctness 81
7.3 Selective-censorship resistance 84
7.4 Proposal hiding 86
7.5 Accountable equivocation 87
7.6 What the design gives 89

8 The implementation 91
8.1 The workspace 92
8.2 The runtime 93
8.3 The life of a slot 94
8.4 Cryptographic choices 97
8.5 Execution on LiteSVM 99
8.6 Transport 100
8.7 The store 101
8.8 The gateway 101
8.9 The CLI 102
8.10 Genesis 103
8.11 The API and the contract 104
8.12 Build and test 105

9 Acrux Net, measured 106
9.1 The net 107
9.2 How the net measures itself 109
9.3 Six hundred seconds 109

9.4 A slot, measured 110
9.5 Window handovers 111
9.6 One state on twelve validators 113
9.7 What a send costs 113
9.8 One proposer down 114
9.9 Six hundred seconds on the server 115

10 The Fault Lab 117

10.1 The runner 118
10.2 Reading a run by its attestations 120
10.3 Last look 121
10.4 Censorship 122
10.5 Crash 123
10.6 Equivocation 123
10.7 Split commitments 124
10.8 Private front-run 125
10.9 Six runs, six proofs 126

11 The Hub Registry 128

11.1 Why hubs live on chain 129
11.2 One program on two chains 129
11.3 Accounts 130
11.4 Instructions 132
11.5 Flows 134
11.6 The indexer 135
11.7 From the registry to Algorithm 3 136

12 The SDK 137

12.1 Install and connect 138
12.2 Sending a transaction 138
12.3 The lifecycle of a transaction 142
12.4 Typed errors 144
12.5 What a send costs 145
12.6 The contract types 146

13 Operating a node and a hub 147

13.1 What changes for a validator 148
13.2 Install 148
13.3 A local three-hub net 149
13.4 Configuration 151
13.5 Telemetry 152
13.6 Following Acrux Net 153
13.7 Registering and joining a hub 155

13.8 Ports and the public host 157

14 ACRUX 159

14.1 Gas and stake 160
14.2 Supply and launch 160
14.3 Where the value comes from 161
14.4 Buy and burn 161
14.5 Fees on Acrux Net 162

15 The road 163

15.1 The goal and how it is checked 164
15.2 Build 1: Solana programs, moved over 165
15.3 Build 2: mainnet replay 166
15.4 Build 3: a bridge from Solana 167
15.5 Build 4: ACRUX as gas and stake 168
15.6 Build 5: operators staking ACRUX 168
15.7 Build 6: a window that survives a split commitment 169
15.8 Build 7: concurrent hubs 170
15.9 The first day 170

Appendices 172

A Parameters 173

A.1 Protocol parameters 173
A.2 Values the paper leaves open 173
A.3 Choices of construction 174

B Interfaces: JSON-RPC, events, HTTP API, CLI 176

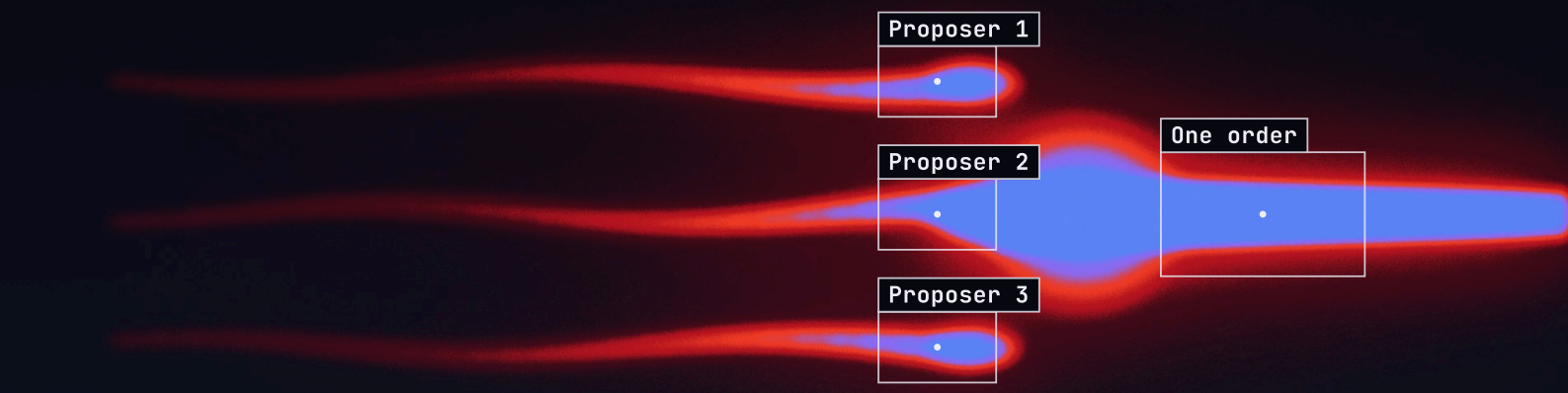
B.1 JSON-RPC 176
B.2 Event stream 179
B.3 Gateway HTTP routes 181
B.4 HTTP API 182
B.5 CLI 185

C Hub Registry reference 187

C.1 On Solana mainnet 187
C.2 Encoding 187
C.3 Accounts 188
C.4 Instructions 190
C.5 Reading the state at an epoch 194
C.6 Events 194
C.7 Errors 195

D Wire formats 197
D.1 Encoding 197
D.2 The envelope 197
D.3 Commitment and attestation 198
D.4 Slice encryption 200
D.5 Shreds and their Merkle proofs 200
D.6 Votes and certificates 202
D.7 Transactions 203
D.8 Threshold fragment 203
D.9 Hashed objects 204
E From the paper to the code 206
E.1 Reading the tables 206
E.2 Algorithm 1 206
E.3 Algorithm 2 207
E.4 Algorithm 3 208
E.5 Algorithm 4 208
E.6 Validity and agreement 209
E.7 Theorem and lemmas 210
E.8 Threshold transactions, fees and the Fault Lab 210
References 212

$\delta_{block}$ 50 MS · SOLANA ABOUT 215 MS
K 3 PROPOSERS PER SLOT · NO LEADER
SOLANA PROGRAMS UNCHANGED ON THE SVM




1

# Solana at 50 ms

Acrux is a faster Solana: its own chain, a fork that runs Solana programs unchanged, with New Constellation as its consensus. Every 50 ms three proposers author the slot together, and none of them gets the last look.

What Acrux is, then why a slot without a leader is the faster and fairer slot: what the single author of a Solana slot can do, what it costs users, the designs that answer it, and the one Acrux runs as its consensus.

## 1.1 A faster Solana

Acrux is a chain. It is a fork of Solana with its own ledger, its own validators and its own token, and it runs Solana programs and Solana transactions unchanged on the Solana Virtual Machine. What it changes is how a slot is made. Solana gives each slot to one leader, and since Alpenglow its mainnet slots measure about 215 ms (212 to 218 ms over ten performance samples on 2026-11-17 UTC) [13]. Acrux replaces the leader with New Constellation, the multiple-concurrent-proposer design for Alpenglow [1]: a slot lasts 50 ms, about four times shorter, and three co-located proposers author it together. Each seals its block under a fresh key, the three exchange signed commitments within $2\delta_{\text{local}}$, and only then are the keys released, so no proposer sees the others' transactions before its own block is fixed. Validators merge every block of the slot into one order by priority fee, execute it, and vote through Alpenglow on consensus blocks of four slots.

| 50 ms | 3 | 324 ms | 2,640 |
|---|---|---|---|
| Slot length; Solana mainnet's measures about 215 ms | Proposers in every slot, none of them a leader | Consensus block finalized, p50, on Acrux Net | Consensus blocks in the 600 s run, every one finalized |

ACRUX is the chain's native token: the gas every transaction pays and the stake every validator holds (Chapter 14). The chain runs now as Acrux Net, with genesis hubs in Frankfurt, New York and Tokyo and their geographic delays on the wire (Chapter 9). Its hubs are registered in the Hub Registry, a program that runs on Acrux itself, where it drives the proposer schedule, and on Solana mainnet, where Solana validators register and join hubs with the identity they already hold (Chapter 11).

> **Solana's speed is a choice its leader schedule made, not a limit of the SVM. Acrux makes the other choice and runs Solana's programs on it, about four times faster.**

The rest of this chapter tells why the leaderless slot is also the fairer one. A slot with one author gives that author the last look at every transaction in it.

## 1.2 One author per slot

Solana divides time into slots and assigns each slot to a validator through a leader schedule computed from stake. During its slots, the leader receives the transactions users send, builds a block from them, and streams it to the rest of the network. Every other validator checks the block and votes on it. Consensus decides whether the block is accepted; it does not decide what the block contains. That decision belongs to the leader alone.

The New Constellation paper names the three powers this gives the leader of the moment [1]. The leader can observe all pending transactions before it commits to anything. It can reorder them, and it can censor any it does not like. And it can insert its own transactions anywhere in the sequence, in particular directly in front of a user's transaction whose effect on a price it has already seen. The paper calls this position a temporary monopoly on transaction inclusion and ordering: temporary because the schedule rotates, a monopoly because within its slot no one else has a say.

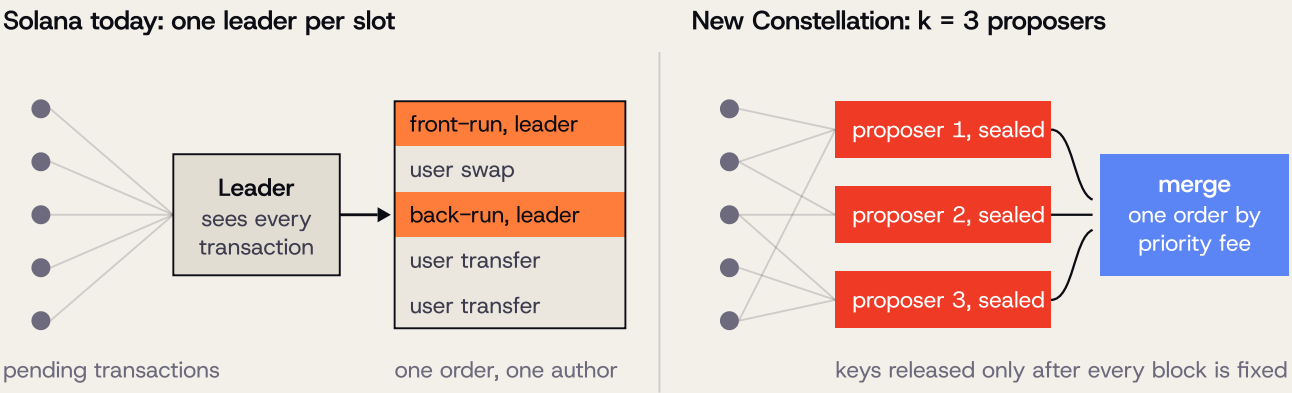



Figure 1.1 Who decides a slot. Left: a single leader receives every pending transaction and fixes the order, here a sandwich around a user swap. Right: in New Constellation the k proposers of a slot build blocks concurrently, each sealed until all are committed, and validators merge them by a rule no proposer controls.

Nothing in this arrangement is specific to a dishonest operator. A leader that follows the protocol to the letter still sees the pending transactions first and still decides their order; the protocol simply gives it no rule to follow. What it does with that freedom is a matter of incentives, and the incentives are measurable.

## 1.3 What the position is worth

The profit available to whoever orders a block is known as maximal extractable value (MEV). Daian et al. introduced the term and documented its forms on Ethereum: front-running, sandwich attacks and priority gas auctions, in which bots bid against each other for position [7]. They also

identified a second harm beyond the direct loss to users: once ordering revenue is large enough, a rational validator has an incentive to deviate from the protocol to capture it, which they called consensus instability.

A sandwich is the plainest case. A user submits a swap with some allowed slippage. The party that orders the block places a buy of the same asset immediately before it, which moves the price against the user, lets the user's swap execute at the worse price, and sells immediately after. The user receives less; the difference is the attacker's. The only requirement is to see the swap before the order is fixed, and the leader of a slot sees everything before the order is fixed.

On Solana the effect has been measured directly. Gerzon et al. studied four months of early 2025 on Jito, the most adopted Solana validator client, and counted 521,903 sandwich attacks, which cost users over $7.7M; they also found users paying a further $2.4M for protections that did little against them [8]. Over 95% of active stake runs the Jito client [9], so the client measured is the one nearly all stake runs.

| 521,903 | $7.7 M | $2.4 M | 95 % |
|---|---|---|---|
| Sandwich attacks on Jito, four months of early 2025 | Cost to users, at least | Paid for protections on top | Active stake on the Jito client, over |

1.3.1

### Redistributing the value, or removing it

The dominant practical answer to MEV has been proposer–builder separation [15]. It accepts that the producer of a block holds the ordering and auctions that right to specialised builders, who compete to construct the most profitable block. The paper's assessment is structural: separation optimises and redistributes the value of ordering, rather than removing the harm to users [1]. Fox, Pai and Resnick make the point formally: a single proposer with exclusive ordering power cannot provide censorship resistance and rational participation at the same time, which is the reason to dilute that power rather than sell it [16].

The economics literature arrived at a related conclusion in traditional markets. Budish, Cramton and Shim showed that continuous limit order books create an arms race in which speed, not information, earns the profit, and proposed frequent batch auctions: orders are collected over a short interval and cleared together, so an advantage smaller than the interval is worth nothing [6]. A slot is such an interval. If the transactions of a slot are merged and ordered by a rule rather than by a party, what counts is the slot a transaction lands in, not its exact arrival time and not who received it.

Garimidi, Neu and Resnick give the corresponding design for blockchains: multiple concurrent proposers (MCP) [5]. Block content is sourced from several proposers at once, so that no single party can censor or reorder all submitted transactions. They show that MCP can achieve selective-censorship resistance and hiding with a single additional round trip over a

standard consensus protocol. The two properties are the ones a leader breaks: a transaction reaches the chain if any one of its proposers is honest, and no proposer can react to what another proposer is carrying.

## 1.4 The motivation New Constellation states

Kniep and Wattenhofer open their paper with the gap between what a blockchain promises and how most of them run: the promise is that no single party is in control, and in practice most blockchains, Solana included, still grant the temporary monopoly described above [1]. Multiple concurrent proposers break it, provided two conditions hold. The proposers of a slot must include transactions in the same slot, and their transactions must be merged, so that inclusion and order are no longer a single party's decision.

New Constellation sets out to do this while keeping what makes Alpenglow, Solana's consensus, fast and simple. The paper lists the properties it offers under three headings. Under MEV resistance: no leader, censorship resistance that needs only one non-censoring proposer of a slot, a deterministic order by priority fee, blocks hidden from the other proposers until all are committed, transactions that can be split so no single proposer can read them, and accountability for late or equivocating proposers. Under performance: transactions included every 50 ms, no extra global hop, Alpenglow's bandwidth, and lateness bounded by a few milliseconds instead of the global timeout. Under robustness: Alpenglow runs unchanged on consensus blocks, validators execute before they vote, nodes need no synchronised clocks, and a crashed proposer loses only its own blocks. Chapter 7 proves the guarantees; the rest of this document shows each one built and measured.

## 1.5 Designs that answer the same problem

New Constellation is not the first design to take block content away from a single party, and the differences between the designs are in the details that decide whether a leader survives in another form.

### 1.5.1 Legacy Constellation

Anza published Constellation in March 2026 and set multiple concurrent proposers as Solana's direction [2, 10]. In that design, which the New Constellation paper calls Legacy Constellation (LegCon), proposers send their transactions to a set of attesters, which timestamp them and forward them to a leader; the leader must assemble each batch from the attested transactions. The result is a protocol-enforced economic tick of 50 ms. It needs an additional layer of attesters and synchronised clocks, and it keeps a leader that, although tightly constrained, still assembles the block. New

Constellation has neither attesters nor a leader: the k proposers of a slot produce Alpenglow blocks themselves, and co-location replaces synchronised clocks [1].

### 1.5.2 Other multiple-proposer designs

Several MCP and censorship-resistant designs appeared in 2025 and 2026, and the paper's Related Work places New Constellation among them [1, 17]. Sei Giga makes proposals available first, orders them by consensus, and executes the blocks asynchronously afterwards [18]. Cadence runs an independent consensus instance per slot, each with several proposers, and reaches short-term censorship resistance and hiding at the fast-path latency of single-leader consensus [19]. Gatling runs several staggered instances of an atomic broadcast protocol and merges their outputs with a deterministic interleaving rule, so the time between proposals can be made arbitrarily small [20]. On Ethereum, fork-choice enforced inclusion lists (FOCIL) let a committee of validators force transactions into blocks, a lighter route to censorship resistance that keeps a single block builder [21].

Two choices separate New Constellation from these. It decides all proposer blocks of a consensus slot together, so validators can execute before they vote and a certificate certifies the resulting state; designs that decide proposer blocks independently must execute after consensus. And it needs no new consensus: one Alpenglow instance runs unchanged on blocks of four slots. In the taxonomy of Marsh, Ranchal-Pedrosa and Resnick, LegCon is a form of constrained assembly, in which several parties restrict a block that one party assembles, while New Constellation is closest to a fixed proposal vector, with k proposals per slot [17].

**Table 1.1** Designs that take block content away from a single party, as the New Constellation paper and its Related Work describe them [1].

| Design | Who assembles block content | What it rests on |
|---|---|---|
| Solana today | One leader per slot | A leader schedule by stake |
| Legacy Constellation | A constrained leader, from attested batches | Attesters, synchronised clocks, a 50 ms tick |
| Sei Giga | Several proposers, ordered by consensus | Execution after consensus |
| Cadence | Several proposers per slot | One consensus instance per slot |
| Gatling | Staggered broadcast instances, interleaved | A deterministic interleaving rule |
| FOCIL (Ethereum) | One builder, bound by inclusion lists | An inclusion-list committee |
| **New Constellation** | **k co-located proposers, merged by rule** | **A $2\delta_{\text{local}}$ hash exchange inside a hub; one Alpenglow instance; execution before voting** |

## 1.6 Why now

New Constellation is built for Alpenglow: its proposers start on Alpenglow's ParentReady event, its blocks ride Alpenglow's Rotor, and its consensus blocks are decided by Alpenglow's Votor [3]. Until Alpenglow ran on Solana, a design built on it was a design for a future network. Alpenglow went live on Solana mainnet on 2026-10-21, in epoch 1064 [4]. The New Constellation paper followed three weeks later, on 2026-11-14 [1], with its algorithms in pseudocode, its proofs and its parameters. A design this complete does not need to wait for a chain to adopt it: Acrux runs it as its own chain, on the SVM, with Solana's programs.

Table 1.2 The sequence of work this document builds on.

| Date | Work | What it established |
|---|---|---|
| 2015 | Budish, Cramton, Shim [6] | Frequent batch auctions end the speed race inside an interval |
| 2020 | Daian et al., *Flash Boys 2.0* [7] | MEV defined; front-running and consensus instability documented |
| 2025 | Kniep, Sliwinski, Wattenhofer, *Alpenglow* [3] | Votor and Rotor, Solana's new consensus |
| 2025-09 | Garimidi, Neu, Resnick [5] | MCP: selective-censorship resistance and hiding in one extra round trip |
| 2025-10 | Gerzon et al., IMC '25 [8] | 521,903 sandwiches on Jito in four months, over $7.7M to users |
| 2026-03 | Anza, *Constellation* v0.9 [2] | MCP set as Solana's direction, with attesters and a constrained leader |
| 2026-10-21 | Alpenglow live on mainnet, epoch 1064 [4] | The consensus New Constellation runs on |
| 2026-11-14 | Kniep, Wattenhofer, *New Constellation* v1.0 [1] | k proposers per 50 ms slot, no leader, no attesters |
| 2026-11-16 | Acrux, this whitepaper | A Solana fork with New Constellation as its consensus, live on three hubs |

## 1.7 The insight

A slot needs a single author only if the participants cannot agree, within the slot, on what each of them has fixed. Across the globe that agreement costs a round of messages bounded by $\delta_{\text{global}}$, which the paper sets at 250 ms, five times the length of a slot. Inside one city it costs far less. If the proposers of a slot sit within a small delay $\delta_{\text{local}}$ of each other, they can exchange signed hashes of their blocks within $2\delta_{\text{local}}$, which is 20 ms with the paper's 10 ms. That exchange does the two things a leader is otherwise needed for. It fixes, for every proposer, the set of blocks that make up the slot, so a hash exchange replaces the leader's authority. And because the blocks travel

encrypted and each proposer releases its key only after the exchange, no proposer learns another's transactions before its own block is fixed, so a commit–and–reveal replaces trust in the leader.

> **The single author of a slot is a property of the protocol, not of physics: proposers a few milliseconds apart can fix a slot together, in less time than the slot lasts.**

The idea asks one thing of geography: proposers must be co–located, which New Constellation organises as hubs: groups of validators in one place, registered on chain, between which the windows of the schedule rotate. Solana's validators already cluster in a few data–centre cities. With a generous $\delta_{\text{local}}$ of 25 ms and a minimum hub stake of 1%, the paper finds only 1.2% of today's stake outside a place that could form an active hub [1]. Chapter 6 describes hubs and the schedule built on them.

---

## 1.8 New Constellation in one slot

Time runs in slots of $\delta_{\text{block}}$ = 50 ms, grouped into windows of $w$ = 16 slots. The schedule assigns every window to one hub and draws $k$ = 3 proposers from its members. All three build a block in every slot of the window, and every slot runs the same sequence. The names below are the ones Acrux uses on its surfaces; the mechanism is Section 1.3 of the paper [1].

1. **Seal.** On its ParentReady, each proposer draws a fresh symmetric key $K_s$, builds its block from the transactions users sent to it, and streams it out slice by slice, every slice encrypted under $K_s$ and disseminated to all validators through Rotor. The co–proposers' ParentReady events differ by at most $\delta_{\text{local}}$.
2. **Commit.** With its last slice, each proposer sends a signed hash of its whole encrypted block, which also binds $H(K_s)$, directly to its co–proposers: `Commit(s, q, h)`.
3. **Attest.** Each proposer waits until the commitments of all co–proposers arrive, or until $2\delta_{\text{local}}$ have passed from the start of its exchange, and then sends a signed attestation to all validators, naming which co–proposers were on time and to what they committed. The attestation carries its key: `Attestation(s, q, π, K)`.
4. **Reveal.** A validator accepts the slot on a set $P$ when every attestation it holds agrees, $P$ is a strict majority and no proposer equivocated. It decrypts the blocks of $P$ with the released keys.
5. **Merge.** The transactions of all accepted blocks fall into one deterministic order by priority fee, and the resulting execution block is executed.
6. **Vote.** Every four slots form a consensus block, and validators vote once on its post–execution state through Alpenglow.

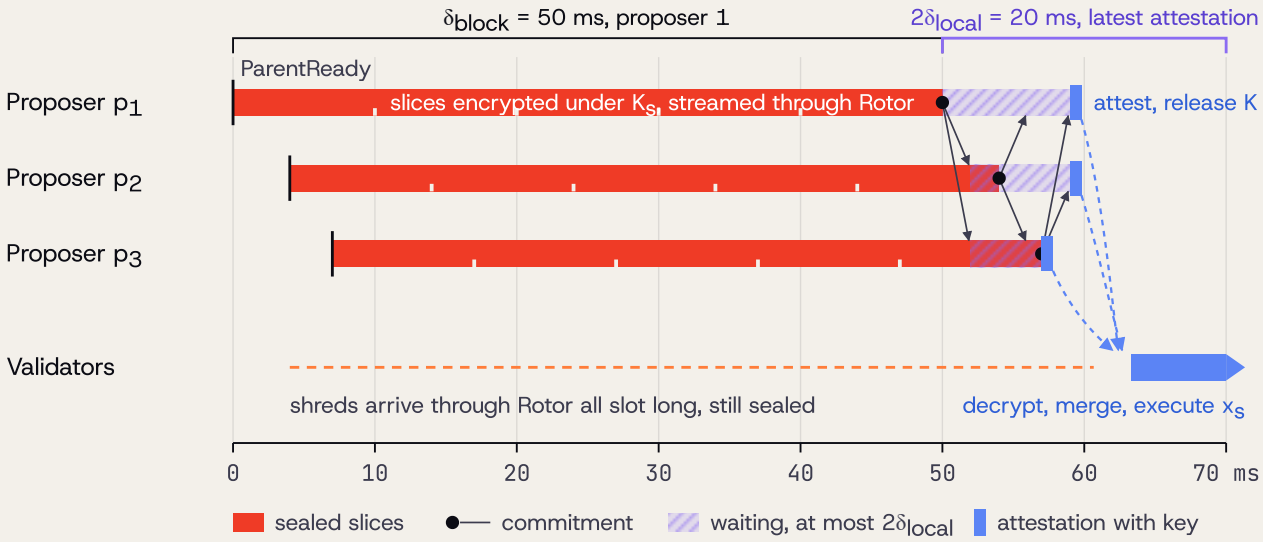



**Figure 1.2**

One slot of a window with k = 3 proposers, drawn to scale on the paper's parameters. The proposers' ParentReady events fall at 0, 4 and 7 ms, within $\delta_{\text{local}}$; each builds for 50 ms and commits at 50, 54 and 57 ms; commitments take 2 ms inside the hub. Each proposer's exchange starts at its own send or at the first commitment it receives, whichever is earlier, and it attests as soon as all commitments are in, here at 59, 59 and 57 ms, at the latest $2\delta_{\text{local}}$ after the start. As in the paper's Figure 2, the transit to validators is drawn short.

Two properties of the slot follow from the figure. First, the attestations and keys reach the validators at about the same time as the blocks. An attestation is delayed by the local exchange, at most $2\delta_{\text{local}}$, and then goes directly to every validator; a block takes Rotor's relay hop, which on a network like Solana mainnet adds a detour of roughly 20% of the global network delay [1]. So the exchange adds no latency a validator waits for. Second, a proposer that wants to see the others before fixing its own block gains nothing by waiting: the others' blocks are ciphertext until their keys are released, the keys are released only in attestations, and an attestation is sent only after its sender's commitment. A proposer whose commitment arrives after $2\delta_{\text{local}}$ is attested as late, and validators do not accept its block. Chapter 3 specifies the proposer and Chapter 4 the validator.

Beyond the slot, the design pipelines. Proposers never wait for votes; they produce their sixteen blocks back to back. Validators decrypt, merge and execute every slot as it arrives and vote once per consensus block of four slots, on the hash of the state after executing it, so a certificate certifies state and not only content. A disagreement between attestations, or anything missing at the timeout, leads to a skip vote, and as in Alpenglow a skipped consensus block skips the rest of its window. A proposer crashed from the start of its window does not cause a skip: all its co-proposers attest it as late, the attestations agree, and the vote covers the other two blocks.

## 1.9 What Acrux is

Acrux is a chain and everything around it. Its node is built from the paper's algorithms line by line, and the chain it runs makes a leaderless slot every 50 ms. It has five parts.

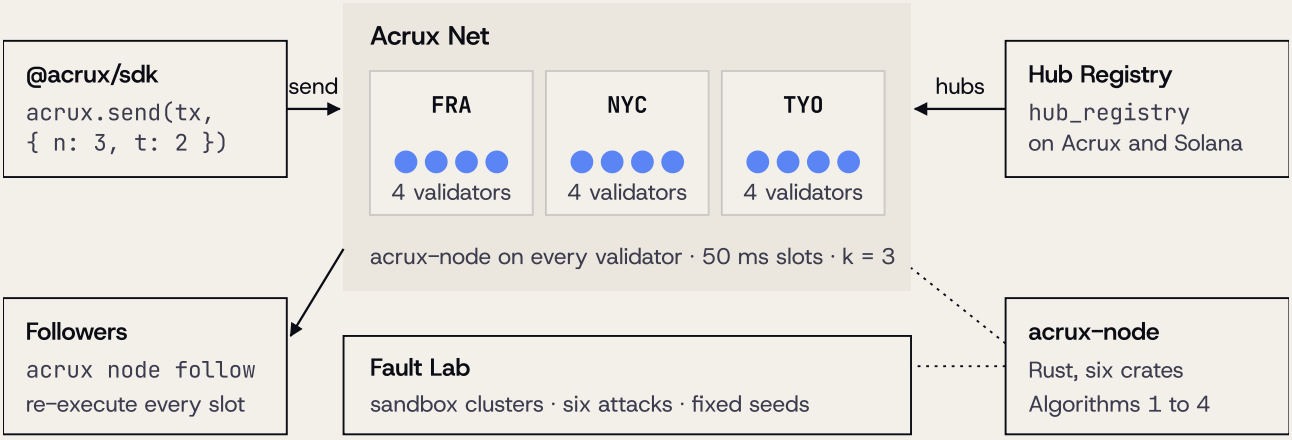



Figure 1.3

The parts of Acrux. The SDK sends to the proposers of Acrux Net; the Hub Registry holds the hubs, on Acrux, where it drives the schedule, and on Solana mainnet; the Fault Lab runs sandbox clusters of the same node; followers re-execute every finalized slot and check its state hash and certificate. Dotted: the same `acrux-node` binary runs each cluster.

| | |
|---|---|
| **acrux-node** | The node, a Rust workspace of six crates. It implements the paper's proposer (Algorithm 1), validator (Algorithm 2), proposer schedule (Algorithm 3) and transaction selection (Algorithm 4), with Rotor dissemination, Alpenglow's Votor on consensus blocks, t-of-n threshold transactions and the fee-payer reserve rule. Execution runs real Solana transactions on the Solana Virtual Machine. Its `acrux` CLI starts a local multi-hub cluster in one command or follows Acrux Net and re-executes every slot. Chapter 8. |
| **Acrux Net** | The chain itself, live on three genesis hubs, FRA, NYC and TYO, with four validators each and the hubs' geographic delays applied on the wire: k = 3 proposers per window, 50 ms slots, windows of 16 slots rotating between hubs, Alpenglow voting on every fourth slot. Its explorer shows each slot as it happens. Chapter 9. |
| **Fault Lab** | Six attacks New Constellation is built against, each run on a sandbox cluster of twelve validators with a fixed seed and each ending in the outcome the paper proves, with its evidence: a proposer that |

| | |
|---|---|
| | waits for a last look, a censoring pair, a crashed proposer, an equivocator, split commitments and a front-run on a private swap. Chapter 10. |
| **Hub Registry** | The program `hub_registry`, where hubs are registered by code and location with a deposit and validators join them by identity, as the paper's Section 3.1 makes them on-chain transactions. It runs on Acrux, where its hubs and memberships drive the chain's proposer schedule, and on Solana mainnet, where Solana validators register and join hubs with the identity they already hold. Chapter 11. |
| **@acrux/sdk** | The developer surface. `await acrux.send(tx, { n: 3, t: 2, signer })` sends a transaction to three proposers of its target slot, split so that any two rebuild it and no single one can read it. Chapter 12. |

### 1.9.1 What Acrux decides

The paper specifies the protocol and its proofs, and names functions where it does not need their construction: a hash, a symmetric cipher, an erasure code, a threshold scheme, a transport. An implementation has to fix each one, together with the byte layout of every message and several values the paper leaves open. Acrux fixes them as below; Appendix A holds every value and Appendix D every wire format.

**Table 1.3** Choices the paper leaves to an implementation, as acrux-node makes them.

| Open in the paper | Acrux | Told in |
|---|---|---|
| Block cipher and key binding | ChaCha20-Poly1305, one fresh key per block, $H(K)$ in the header so the commitment binds the key | Chapter 8 |
| Erasure code | Reed-Solomon for shreds and fragments, 64 shreds per slice of which any 32 rebuild it | Appendix D |
| Threshold scheme | Shamir key shares over GF(256) with erasure-coded ciphertext, after Krawczyk [11] | Chapter 5 |
| Wire formats | Byte layouts of commitments, attestations, shreds and fragments | Appendix D |
| Transport | QUIC, one endpoint per validator | Chapter 8 |
| Hash exchange redundancy | Every commitment sent several times; no forwarding round, so the window stays $2\delta_{\text{local}}$ | Chapter 3 |
| Rotor relays for k proposers | Sampled per slot, per slice index and per co-proposer | Chapter 3 |
| Base timeout $\Delta_{\text{timeout}}$ | 750 ms, the $3\delta_{\text{global}}$ that assumption (T) allows | Chapter 2 |
| Slot capacity | $m = 4$ tracks of 2,000,000 compute units | Chapter 5 |
| Fees | Base fee 5,000 lamports per signature, half burned; 1,000 lamports per included copy or fragment | Chapter 5 |
| Window ordering | SIMD-0675 regional runs [12] | Chapter 6 |

| Open in the paper | Acrux | Told in |
|---|---|---|
| Active hubs | At least 4 members and 1% of total stake | Chapter 6 |
| Execution | The Solana Virtual Machine | Chapter 8 |

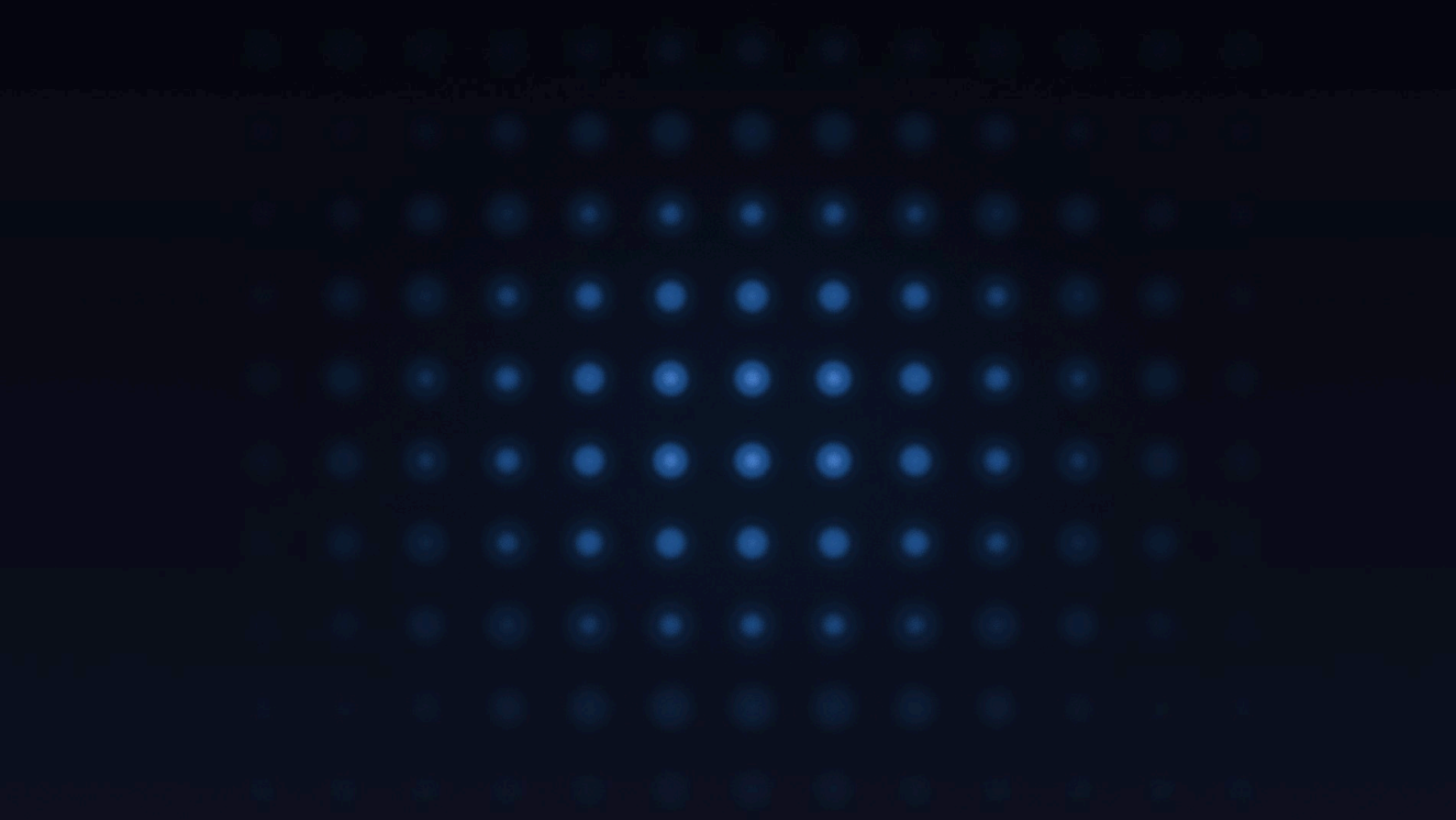
On Acrux Net, ten minutes of fault-free running on the twelve validators at fifty transfers a second finalized every one of the 2,640 consensus blocks of the run, with no skips and no mismatch in 10,564 state-hash checks across validators. Chapter 9 reports the run in full.

## 1.10 The rest of this document

The protocol as specified comes first, then its proof, then each part of Acrux as built and measured.

Table 1.4 Where each subject is told.

| Chapter | Subject |
|---|---|
| Chapter 2 | Model and assumptions: validators, stake, the adversary, delays, cryptography, notation |
| Chapter 3 | Proposing a slot: seal, commit, the $2\delta_{\text{local}}$ exchange, attest (Algorithm 1) |
| Chapter 4 | Validating and voting: validity, agreement, execution, timeouts, cblocks (Algorithm 2) |
| Chapter 5 | Ordering, threshold transactions and fees (Algorithm 4) |
| Chapter 6 | Hubs and the proposer schedule (Algorithm 3) |
| Chapter 7 | Correctness: Theorem 2.1 and Lemmas 2.2 to 2.4 |
| Chapter 8 | The implementation: crates, cryptography, transport, execution |
| Chapter 9 | Acrux Net, measured |
| Chapter 10 | The Fault Lab: six attacks and their outcomes |
| Chapter 11 | The Hub Registry on Acrux and on Solana mainnet |
| Chapter 12 | The SDK |
| Chapter 13 | Operating a node and a hub |
| Chapter 14 | ACRUX, the native token: gas, stake, launch |
| Chapter 15 | The road |
| Appendix A | Parameters |
| Appendix B | Interfaces: JSON-RPC, events, HTTP API, CLI |
| Appendix C | Hub Registry reference |
| Appendix D | Wire formats |
| Appendix E | From the paper to the code: every algorithm line and lemma mapped |



2

# Model and assumptions

Validators, stake, hubs and the network between them: what New Constellation assumes, what Alpenglow already gives it, and the names used for each.

The participants, the faults they may have, the network between them and the cryptography they rely on, as New Constellation assumes them; then the units of time, the roles, the parts of Alpenglow the protocol uses, and every symbol the rest of this document writes.

## 2.1 Validators and stake

The system is a set of validators, each identified by a signing key and holding stake. Stake is the unit of every threshold in the protocol: a certificate needs a share of the total stake, not a count of validators. Following the paper's Section 3.1, $\sigma_i$ denotes the stake of validator $i$ as a fraction of the total [1]. Every validator pays Alpenglow's validator admission ticket (VAT), so running several validators to look independent means paying it several times [3].

Every validator votes. A validator that is a member of an active hub can also be drawn as a proposer; a validator in no hub only votes. How often a member is drawn is set by its proposer stake $\sigma^{\wedge}_i$, its stake capped so that no member exceeds $1/k$ of its hub, and the windows of an epoch are divided among hubs by hub stake $\sigma^{\wedge}_h$. The excess stake of a capped member still counts for voting and validator rewards. Chapter 6 defines these quantities and the schedule built from them; this chapter needs only that the schedule is a deterministic function of a stake snapshot and the hub memberships, seeded from the epoch, so that every validator computes the same proposers for every window.

## 2.2 The adversary

New Constellation adopts Alpenglow's model unchanged [1, 3]. Less than 20% of the stake is byzantine: it may deviate from the protocol arbitrarily, collude, delay or withhold messages and sign anything it holds keys for. Up to another 20% may be crashed or offline: it follows the protocol while it runs and sends nothing otherwise. Equivocation, signing two different messages where the protocol allows one, is detectable, because both signed messages exist and either can be shown to any validator.

Table 2.1 The stake thresholds of the model. Byzantine stake is bounded from above; the remaining thresholds are the correct and online stake each guarantee needs [1].

| Quantity | Bound | What it buys |
|---|---:|---|
| Byzantine stake $f$ | below 20% | Safety: a single finalized chain, every certified state hash correct |
| Crashed or offline stake | up to another 20% | Tolerated with liveness intact |
| Correct and online stake | at least 60% | Liveness after GST; notarization and skip certificates |
| Correct and online stake | at least 80% | Fast finalization in one round of voting |

The model also bounds what one window's proposers can do. A proposer is a validator, so a byzantine proposer counts against the 20%; it may withhold its commitment, send it to some co-proposers only, attest falsely, release no key, or sign two blocks. The protocol's answer to each is a skip, a late mark or a slashing proof, never a corrupted state (Chapter 7). For liveness the paper distinguishes a crashed proposer, one that sends no messages at all in the window. A window finalizes all its consensus slots as long as more than $k/2$ of its proposers are correct and the others are crashed. With $k = 3$ a window thus survives one crashed proposer; with $k = 2$ a crash leaves no majority, which is why the paper proposes $k = 3$ as the smallest number that tolerates one.

## 2.3 Network and time

### 2.3.1 Partial synchrony

The network is partially synchronous. Safety holds even during periods of asynchrony, when messages may be delayed arbitrarily; liveness holds during long enough periods of synchrony. The global stabilisation time (GST) marks the point after which the network behaves: after GST, a message between any two correct nodes arrives within $\delta_{\text{global}}$, and a message between two correct co-located nodes arrives within the much smaller $\delta_{\text{local}}$ [1].

With $\delta_{\text{local}} = 10$ ms, $2\delta_{\text{local}} = 20$ ms is two fifths of a slot.

Two delays carry the whole design. $\delta_{\text{global}} = 250$ ms bounds every message across the network and sets Alpenglow's timeouts. $\delta_{\text{local}} = 10$ ms bounds every message between the members of a hub, and sets the window of the hash exchange, $2\delta_{\text{local}}$. A third duration, $\delta_{\text{block}} = 50$ ms, is not a network delay but the length of a slot, chosen so that a proposer streams at least one slice per slot.

**$\delta_{\text{local}}$** = 10 ms between any two correct members of one hub
**$\delta_{\text{global}}$** = 250 ms between any two correct nodes, after GST

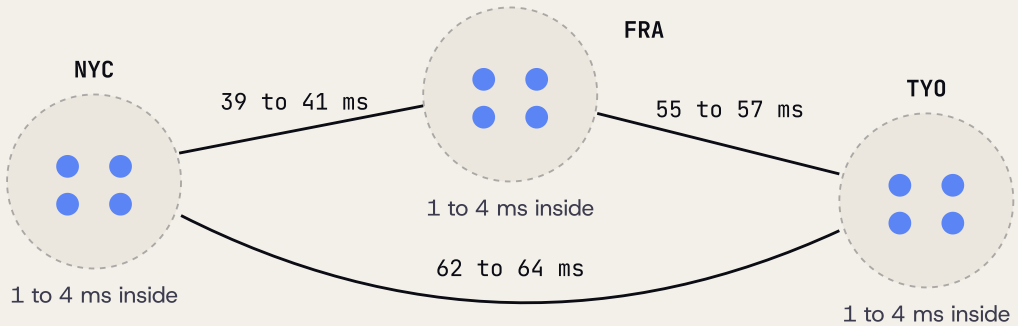



**Figure 2.1** The network model as Acrux Net instantiates it. Members of one hub reach each other within $\delta_{\text{local}}$; any two nodes reach each other within $\delta_{\text{global}}$ after GST. The figures on the links are the one-way delays Acrux Net applies on the wire between its genesis hubs, and 1 to 4 ms inside each hub (Chapter 9).

### 2.3.2 No synchronised clocks

New Constellation, like Alpenglow, needs no physical clock synchronisation. Every timeout is measured with the local clock of the node that sets it and is started by a local event, such as the arrival of a protocol message or the node's own emission of ParentReady. This includes the $2\delta_{\text{local}}$ window of the hash exchange, which every proposer measures with its own clock from `start[s]`, the earlier of the moment it sends its own commitment and the moment it receives the first commitment of a co-proposer for slot $s$. The only assumption on clocks is that they run at about the same rate: over a window, their drift is small enough to be absorbed by the slack in $\delta_{\text{local}}$ and $\delta_{\text{global}}$. The proofs ignore drift within a window for the same reason.

Co-location replaces the synchronised clocks of Legacy Constellation. Two proposers whose ParentReady events differ by at most $\delta_{\text{local}}$ complete their blocks at most $\delta_{\text{local}}$ apart, and a commitment takes at most $\delta_{\text{local}}$ to cross; measured from its own sending time, a proposer therefore receives the commitment of a correct co-proposer within $2\delta_{\text{local}}$. No common clock is involved, only two bounded delays inside one city.

### 2.3.3 Geo-distribution

The model adds one assumption to Alpenglow's: validators form local hubs, for example in one city, with a network delay of at most $\delta_{\text{local}}$ between any two correct members, and the proposers of a window are always drawn from one hub. A hub's condition on delay is not checked by any registry. It is enforced by the attestations themselves and is individually verifiable: a fast hash exchange proves to one validator that the other is close, since no message travels faster than light [1]. A member that cannot keep up is marked late by its co-proposers and loses its blocks, without slowing the others.

## 2.4 The timing assumption

The paper's proofs rest on five assumptions. Two are Alpenglow's own guarantees, treated as a black box; one bounds the adversary; one requires determinism; and one, (T), ties the parameters to the network [1].

**Table 2.2** The assumptions of the paper's Section 2.6, under which Theorem 2.1 and Lemmas 2.2 to 2.4 are proved.

| Assumption | Statement |
|---|---|
| (M) Model | Byzantine stake $f$ is below 20%. After GST, at least 60% of the stake is correct and online, messages between correct nodes take at most $\delta_{\text{global}}$, and messages from a correct proposer reach its co-proposers within $\delta_{\text{local}}$. |
| (D) Determinism | Merging, transaction ordering and execution are deterministic. |
| (T) Timing | $\Delta_{\text{timeout}} \geq 3\delta_{\text{global}}$ and $2\delta_{\text{local}} \leq \delta_{\text{global}}$. After GST, a block of a correct proposer reaches all correct validators through Rotor within $2\delta_{\text{global}}$ of its completion, and a correct validator executes a slot within $\delta_{\text{block}}$ once it holds the slot's blocks, its attestations and the state it builds on. |

| Assumption | Statement |
|---|---|
| (AS) Alpenglow safety | If correct nodes follow Alpenglow's voting rules, the finalized blocks form a single chain, and the finalized chains of correct validators are prefixes of one another. This holds for every block-validity predicate, whenever correct nodes cast their first votes, and also if they omit fallback votes. |
| (AL) Alpenglow liveness | After GST: if every correct validator eventually votes notarize or skip in every slot, every slot obtains a notar-fallback or skip certificate at every correct node; and if the same valid block reaches all correct nodes before the timeouts, it is finalized. |

The two inequalities of (T) have direct readings. The first, $\Delta_{\text{timeout}} \geq 3\delta_{\text{global}}$, comes from the latest moment a validator can hold a slot of a correct window. Let $t_v$ be the time validator $v$ emits ParentReady for the window. A correct proposer's ParentReady is at most $\delta_{\text{global}}$ later; it completes its block of the $m$-th slot $m\delta_{\text{block}}$ after that; Rotor delivers the block within $2\delta_{\text{global}}$, and the attestation, sent at most $2\delta_{\text{local}}$ after completion and directly, within $2\delta_{\text{local}} + \delta_{\text{global}}$. So $v$ holds the blocks and attestations of the $m$-th slot by $t_v + m\delta_{\text{block}} + 3\delta_{\text{global}}$, and has executed it $\delta_{\text{block}}$ later. The timeout of a consensus slot must not expire before that.

The second, $2\delta_{\text{local}} \leq \delta_{\text{global}}$, is what makes the hash exchange free. An attestation waits up to $2\delta_{\text{local}}$ for the exchange and then travels one hop; a block travels two hops through Rotor. As long as the exchange is no longer than one global hop, the attestation arrives no later than the block it releases.

> **Acrux sets $\Delta_{\text{timeout}}$ to 750 ms, exactly $3\delta_{\text{global}}$: the smallest base timeout assumption (T) allows.**

The timeout of consensus slot $i$ of a window whose first consensus slot is $i_0$ expires

```
T(i) = Δtimeout + (i - i0 + 1) · δcblock + δblock
δcblock = κ · δblock
```

after the validator's ParentReady. Compared with Alpenglow, only $\delta_{\text{block}}$ is added, for execution; the $2\delta_{\text{local}}$ of the exchange comes free by the second inequality. On Acrux Net the four timeouts of a window fall at 1,000, 1,200, 1,400 and 1,600 ms. A second deadline bounds how long a validator holds back its first vote of a window while an attestation is missing: $\delta_{\text{block}} + 2\delta_{\text{global}} + 2\delta_{\text{local}}$ = 570 ms after ParentReady, by which the attestation of every correct proposer for the first slot has arrived (Chapter 4).

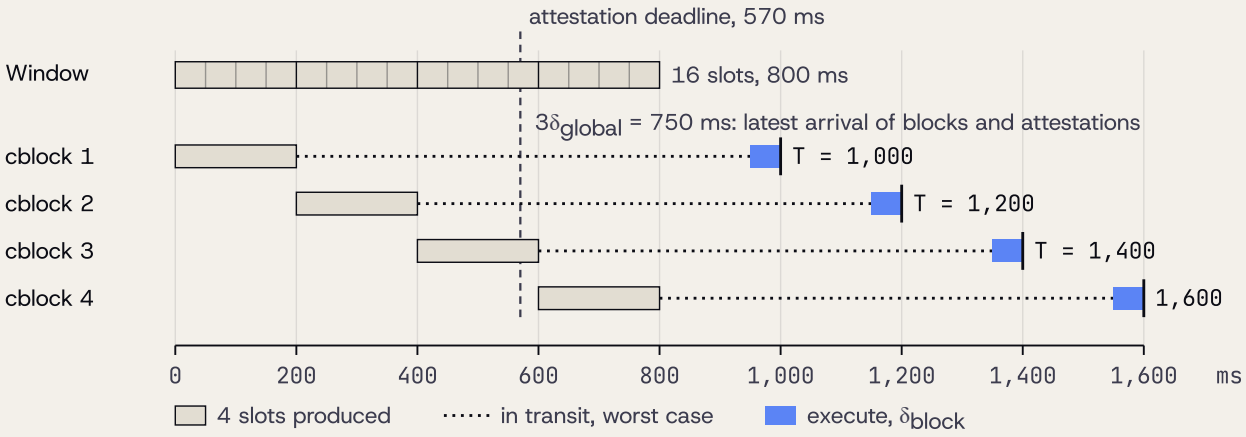



**Figure 2.2** Timeouts of one window on Acrux Net, from a validator's ParentReady. Each cblock's slots are produced in 200 ms; in the worst case allowed after GST their blocks and attestations arrive $3\delta_{global}$ later and execution takes $\delta_{block}$, which ends exactly at the cblock's timeout $T(i)$. With $\Delta_{timeout} = 3\delta_{global}$ the bound of (T) is met with equality.

**Table 2.3** The conditions on the parameters, checked with the values Acrux Net runs.

| Condition | Why | Acrux Net |
|---|---|---|
| $\Delta_{timeout} \geq 3\delta_{global}$ | A correct window's cblocks execute before their timeouts | 750 ms ≥ 750 ms |
| $2\delta_{local} \leq \delta_{global}$ | Attestations arrive no later than the blocks they release | 20 ms ≤ 250 ms |
| $w\delta_{block} \geq \delta_{block} + 2\delta_{global} + 2\delta_{local}$ | A window with an offline proposer finalizes its last cblock no later than a correct one | 800 ms ≥ 570 ms |
| $\lvert P_s \rvert > k/2$ | Agreement needs a strict majority of proposers | at least 2 of 3 |

## 2.5 Cryptographic assumptions

The paper names the functions it needs and the property each must have, and leaves their construction to the implementation. Its proofs use four properties. A commitment is binding: once `Commit(s, q, h)` is sent, the proposer cannot open it to a different block, and because the commitment covers the ciphertext and $H(K)$, it cannot later reveal a key that decrypts the same ciphertext to a different payload. Signatures are unforgeable: nobody can sign in the name of a correct proposer, so attestations and commitments are attributable and equivocation produces a proof. Encryption hides: until the key is released, holding a whole encrypted block reveals nothing about its payload. And erasure coding with Merkle roots lets a validator check every shred against the root its proposer signed and rebuild a slice from any $\gamma$ of its $\Gamma$ shreds. Threshold transactions add a fifth: a computational secret-sharing scheme in the style of Krawczyk, in which fewer than $t$ fragments reveal nothing of the transaction [11].

**Table 2.4** Each function the protocol assumes, the property the proofs use, and the construction acrux-node fixes (Appendix A).

| Function | Property used | Where it matters | Acrux |
|---|---|---|---|
| Hash $h$, $H$ | Binding, collision resistance | Commitments, cblock hashes, state hashes | SHA-256 |
| Signature | Unforgeability | Commitments, attestations, shreds, votes; slashing proofs | Ed25519 |
| Symmetric cipher | Hiding until the key is released | Proposal hiding (Lemma 2.3) | ChaCha20-Poly1305 |
| Erasure code, Merkle tree | Any $\gamma$ of $\Gamma$ shreds rebuild a slice; each shred verifiable | Rotor dissemination, block validity | Reed-Solomon, 32 of 64 |
| Secret sharing | Fewer than $t$ fragments reveal nothing | t-of-n threshold transactions | Shamir over GF(256) |

Assumption (D) completes the list: merging, ordering and execution are deterministic functions of the agreed blocks and the parent state. Every correct validator that executes the same chain of cblocks therefore computes the same state hash, which is what lets a vote, and so a certificate, certify state. Acrux executes on the Solana Virtual Machine with a deterministic virtual schedule of execution tracks (Chapter 5).

## 2.6 Slots, windows and consensus slots

Time is divided into slots $s = 0, 1, 2, \ldots$ of $\delta_{\text{block}} = 50$ ms each, grouped into windows of $w = 16$ slots. The schedule assigns every window to one active hub and draws $k$ co-located proposers $p_1, \ldots, p_k$ from its members. In every slot $s$ of its window, proposer $p_q$ builds one proposer block $b_{s,q}$.

Alpenglow does not see slots. It runs unchanged on consensus slots $i = 0, 1, 2, \ldots$, numbered as Alpenglow numbers its own slots, each made of $\kappa = 4$ consecutive slots: consensus slot $i$ consists of slots $\kappa i, \ldots, \kappa i + \kappa - 1$. A window of 16 slots is therefore an Alpenglow leader window of $w/\kappa = 4$ consensus slots, and it lasts 800 ms, the length of a leader window on Solana today. Whenever this document uses Alpenglow's blocks, votes, certificates and ParentReady, they refer to cblocks and consensus slots; a plain slot always means a 50 ms slot.

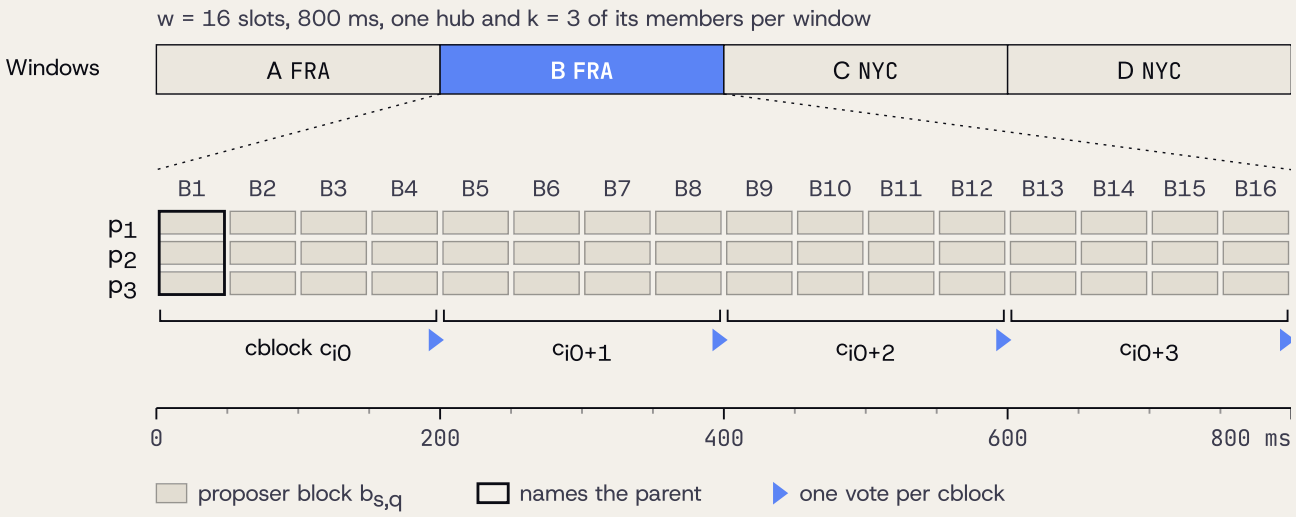



**Figure 2.3** The time structure. Windows of $w = 16$ slots are assigned to hubs, here a run of two windows in FRA and two in NYC as SIMD-0675 orders them into regional runs. Window B expands into 16 slots of $\delta_{block} = 50$ ms, each with $k = 3$ proposer blocks, and into four consensus slots of $\kappa = 4$ slots, one cblock and one Alpenglow vote each. Only the blocks of the window's first slot name a parent.

2.6.1

### Proposer blocks, xblocks and cblocks

Three kinds of block nest inside a window. A **proposer block** $b_{s,q}$ consists of a header, holding the slot $s$ and the proposer index $q$, and a payload of transactions encrypted under a fresh key $K_{s,q}$. It is streamed out in slices throughout its slot; all slices of a block share the key. In the window's first slot, the header also names the parent cblock, sent with the last slice.

The accepted proposer blocks of slot $s$ form its **execution block** $x_s = (b_{s,q})_{q \in P_s}$, where $P_s \subseteq \{1, \ldots, k\}$ is the set of accepted proposers, a strict majority and normally all of them. The xblocks are merged and executed in order, one every $\delta_{block}$. The **consensus block** of consensus slot $i$ is

```
c_i = (i_parent, h(c_parent), x_κi, …, x_κi+κ-1, h_state)
    = (i - 1, h(c_i-1), x_4i, x_4i+1, x_4i+2, x_4i+3, h_state)
for κ = 4 inside a window
```

where `h_state` is the hash of the state after executing $c_i$. No proposer builds a cblock: each validator assembles it from what it validated and executed, and votes on its hash $h(c_i)$, which binds the parent, the agreed sets and attested commitments of all four slots, and the resulting state. Only the first cblock $c_{i0}$ of a window chooses its parent, the one named most often by the accepted blocks of slot $\kappa i_0$, ties going to the lower proposer index; every later cblock has $c_{i-1}$ as its parent, so the cblocks of a window form a hash chain. One Alpenglow decision covers up to $\kappa k = 12$ proposer blocks.

## 2.7 Hubs, proposers and co-proposers

A **hub** is a group of co-located validators, registered on chain under a unique human-readable code, such as an airport code, and a location given as an integer ECEF coordinate triple, as in SIMD-0674 [14]. A hub is active when it has at least $n_{min}$ members, its hub stake reaches $\rho_{min}$, and its members reach each other within $\delta_{local}$; Acrux sets $n_{min} = 4$ and $\rho_{min} = 1\%$ of total stake. Joins, leaves and moves take effect at an epoch boundary announced in advance, with a cooldown.

The **proposers** of a window are $k$ distinct members of its hub, drawn by systematic sampling on a line of proposer stake and given indices 1 to $k$ at random. They are equals: no index has a privilege beyond breaking ties in the parent choice and in fee remainders. Seen from one proposer, the others are its **co-proposers**. Every validator, proposer or not, also runs the validator's loop of Algorithm 2: it rebuilds blocks, collects attestations, decides agreement, executes and votes.

## 2.8 Alpenglow as New Constellation uses it

Alpenglow has two parts: Votor, which votes on blocks and forms certificates, and Rotor, which disseminates blocks [3]. New Constellation treats Alpenglow almost as a black box. It changes two things only, the validity predicate of a block, which becomes Algorithm 2's voting rule, and the timeouts, which leave $\delta_{block}$ for execution [1].

### 2.8.1 Votor on cblocks

For Votor, $c_i$ is simply the block of consensus slot $i$, identified by $h(c_i)$. Algorithm 2 emits the Block event for a cblock at most once per consensus slot, and Votor then casts a notarization vote `NotarVote(i, h(c_i))`; if the slot is not agreed and executed before $T(i)$, the validator casts `SkipVote(i)`. Certificates form at Alpenglow's thresholds of stake.

**Table 2.5** Alpenglow's votes and certificates, as they apply to cblocks [3].

| Certificate | Threshold | Effect |
|---|---|---|
| Fast finalization | 80% | Notarization votes for $h(c_i)$: finalized in one round |
| Notarization | 60% | Notarization votes for $h(c_i)$: notarized, a finalization vote follows |
| Finalization | 60% | Finalization votes: finalized on the slow path |
| Notar-fallback | 60% | Notarization and notar-fallback votes: certified after a split first round |
| Skip | 60% | Skip votes: consensus slot $i$ skipped, and with it the rest of the window |

**ParentReady** is the Alpenglow event that a window may start on a given parent: it is emitted for the first consensus slot of a window once some earlier cblock has a notar–fallback certificate and every consensus slot after it has a skip certificate. In New Constellation it starts the proposers of the next window (Algorithm 1) and sets every validator's timeouts and attestation deadline for the window (Algorithm 2). Co-located proposers emit it at most $\delta_{\text{local}}$ apart, because each forwards the certificates that triggered it directly to its co-proposers as part of regular Alpenglow operation.

### 2.8.2 Rotor

Rotor carries every proposer block, unchanged from Alpenglow. A slice is erasure coded into $\Gamma = 64$ shreds, any $\gamma = 32$ of which rebuild it; with shreds of about 1.2 KB a slice carries about 38 KB, so at least one slice goes out per 50 ms slot. The proposer sends each shred to a relay, and the relays forward it to every validator: two hops. With $k$ proposers per slot, relays are sampled separately for each slot, each slice index and each co-proposer, and since the proposers share the capacity of a slot, relays and validators carry the same load as with one Alpenglow block producer of the same throughput [1]. Commitments and attestations do not use Rotor. A commitment goes directly from a proposer to its $k - 1$ co-proposers; an attestation goes directly to every validator, one hop.

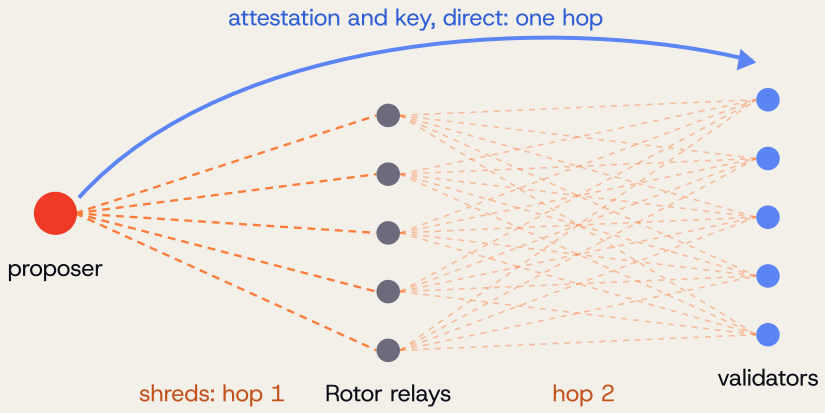



**Figure 2.4** Two paths to a validator. Shreds of a sealed block take two hops through Rotor relays; the attestation that releases the block's key takes one, after at most $2\delta_{\text{local}}$ of exchange. With $2\delta_{\text{local}} \le \delta_{\text{global}}$ the key arrives no later than the block.

## 2.9 Notation

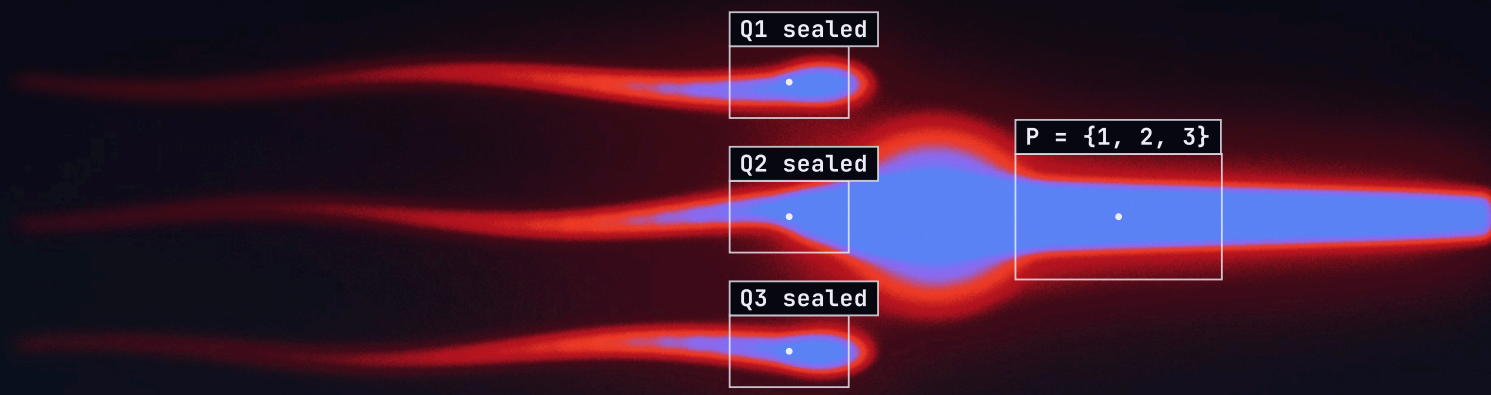
The symbols below are used throughout this document in the form Acrux's specification, docs and interfaces use. Where the paper writes a symbol differently, its letter is in the last column; the meaning is the paper's in every case [1].

**Table 2.6** Every symbol and object of the protocol, in the order the chapters meet them.

| Symbol | Meaning | Paper |
|---|---|---|
| **Parameters and durations** | | |
| $k$ | Concurrent proposers per window, 3 | |
| $\kappa$ | Slots per consensus slot, 4 | $\nu$ |
| $w$ | Slots per window, 16; a window is $w/\kappa = 4$ consensus slots | |
| $\delta_{\text{block}}$, $\delta_{\text{cblock}}$ | Length of a slot, 50 ms; of a consensus slot, $\kappa\delta_{\text{block}} = 200$ ms | $\Delta_{\text{block}}$, $\Delta_{\text{cblock}}$ |
| $\delta_{\text{local}}$, $\delta_{\text{global}}$ | Maximum delay inside a hub, 10 ms; across the network after GST, 250 ms | $\Delta_{\text{local}}$, $\Delta_{\text{global}}$ |
| $\Delta_{\text{timeout}}$ | Alpenglow's base timeout, 750 ms in Acrux | |
| $T(i)$ | Timeout of consensus slot $i$: $\Delta_{\text{timeout}} + (i - i_0 + 1)\delta_{\text{cblock}} + \delta_{\text{block}}$ after ParentReady | |
| Deadline($i_0$) | End of the wait for missing attestations of a window's first slot: $\delta_{\text{block}} + 2\delta_{\text{global}} + 2\delta_{\text{local}}$ after ParentReady | |
| $\Gamma$, $\gamma$ | Shreds per slice, 64; shreds needed to rebuild a slice, 32 | |
| GST | Global stabilisation time, after which the delay bounds hold | |
| **Time, blocks and messages** | | |
| $s$ | A slot, $s = 0, 1, 2, \ldots$ | |
| $i$, $i_0$ | A consensus slot, made of slots $\kappa i$ to $\kappa i + \kappa - 1$; $i_0$ is a window's first, and $\kappa i_0$ the window's first slot | |
| $p_q$, $q$, $p$ | The proposer with index $q \in \{1, \ldots, k\}$ of the window; $p$ is the proposer running Algorithm 1 | |
| $b_{s,q}$ | The proposer block of $p_q$ for slot $s$: header and encrypted payload | |
| $K_{s,q}$, $K_s$ | The fresh symmetric key of $b_{s,q}$; in Algorithm 1, the proposer's own key for slot $s$ | |
| $h(\cdot)$, $H(K)$ | The hash function; the hash of a key, carried in the header so the commitment binds the key | |
| $h_{s,q}$ | The commitment $h(b_{s,q})$: the hash of the block as disseminated, ciphertext included | |
| `Commit(s, q, h)` | The signed message carrying a commitment to the co-proposers | |
| `Attestation(s, q, π, K)` | The signed message of $p_q$ to all validators: its vector and its own block key | |
| $\pi$, $\pi[j]$ | The attestation vector: $\pi[j] = h_{s,j}$ if $p_j$'s commitment arrived on time, $\perp$ otherwise; two attestations agree when their vectors are equal | $\tau$ |
| $P(\pi)$ | The proposers attested by $\pi$: $\{j : \pi[j] \neq \perp\}$ | $P(\tau)$ |
| $P_s$, $P$ | The agreed set of slot $s$, a strict majority of proposers; in Algorithm 1, $P$ is the window's proposer set | |
| $x_s$ | The xblock of slot $s$: $(b_{s,q})$ for $q \in P_s$, merged and executed | |
| $c_i$, $h_{\text{state}}$ | The cblock of consensus slot $i$: parent slot and hash, $\kappa$ xblocks and the hash of the state after executing them | |

| Symbol | Meaning | Paper |
|---|---|---|
| `ParentReady(i, h(c))` | Alpenglow's event that consensus slot $i$ may build on cblock $c$ | |
| `NotarVote, SkipVote` | Alpenglow's votes on $h(c_i)$ and on consensus slot $i$ | |
| **State in the algorithms** | | |
| `start[s]` | Algorithm 1: when the exchange of slot $s$ starts, the earlier of the proposer's own commitment and the first one received | |
| `received[s][j], proposal[s][j]` | Algorithm 1: arrival time and hash of $p_j$'s commitment | |
| `done[s]` | Algorithm 1: whether the attestation for $s$ has been sent | |
| `b[s][q], V[s][q], eq[s][q]` | Algorithm 2: the rebuilt block, the attestation held, and whether $p_q$ equivocated in slot $s$ | |
| `Agreed(s), A` | Algorithm 2: the accepted set of slot $s$, or $\perp$ | |
| $\sigma[s]$ | Algorithm 2: the state after executing slot $s$ | $\Sigma[s]$ |
| `par[i], ĉ` | Algorithm 2: the parent of $c_i$; the parent chosen in a window's first slot | $\pi$ on that line |
| `voted[i]` | Algorithm 2: the set a validator voted for in consensus slot $i$ | |
| $T_q$ | The decrypted transactions of $p_q$'s block | |
| **Stake, hubs and transactions** | | |
| $\sigma_i$, $\hat{\sigma}_i$, $\hat{\sigma}_h$ | Stake of validator $i$ as a fraction of the total; its proposer stake; the stake of hub $h$ | $\rho_i$, $\hat{\rho}_i$, $\hat{\rho}_h$ |
| $c$ | The cap on proposer stake in a hub, so that $\hat{\sigma}_h = kc$ | |
| $H$, $\alpha$ | The set of active hubs; the share of windows assigned by hub stake, 1/2 | |
| $n_{\min}$, $\rho_{\min}$ | Minimum members and minimum hub stake of an active hub, 4 and 1% in Acrux | |
| $f$ | Byzantine stake, below 20% | |
| $t$, $n$ | Threshold and receivers of a t-of-n transaction, $1 \le t \le n \le k$ | |
| $m$, $b_{cu}$ | Execution tracks per slot, 4; compute units per track, 2,000,000 in Acrux | |
| $\rho$ | The fee-payer reserve, 0.001 ACRUX | $\varphi$ |

$\delta_{block}$ 50 MS · $2\delta_{local}$ 20 MS
64 SHREDS PER SLICE · ANY 32 REBUILD IT
COMMIT · ATTEST · REVEAL




3

# Proposing a slot

k proposers build one slot together. Each seals its block under a fresh key, commits to it, waits 2δlocal for the others, and only then releases the key.

In every 50 ms slot of its window, each of the k proposers seals a block under a fresh key, streams it through Rotor, and sends a signed hash of it to its co-proposers. After at most $2\delta_{\text{local}}$ it attests which co-proposers committed on time, and only in that attestation does it release its key.

## 3.1 Blocks and messages

Time runs in slots $s = 0, 1, 2, \ldots$ of $\delta_{\text{block}} = 50$ ms, grouped into windows of $w = 16$ slots. The schedule assigns every window $k = 3$ co-located proposers $p_1, \ldots, p_k$ drawn from one hub (Chapter 6), and in every slot $s$ of the window proposer $p_q$ builds one **proposer block** $b_{s,q}$ [1]. Three larger units are made from these blocks. The accepted proposer blocks of a slot form its **execution block** $x_s = (b_{s,q})_{q \in P_s}$, where $P_s$ is the set of accepted proposers, normally all $k$. Every $\kappa = 4$ consecutive xblocks form a **consensus block** $c_i$, and Alpenglow decides one chain of cblocks [3]. This chapter follows a proposer block from its key to its attestation; Chapter 4 follows what validators do with it.

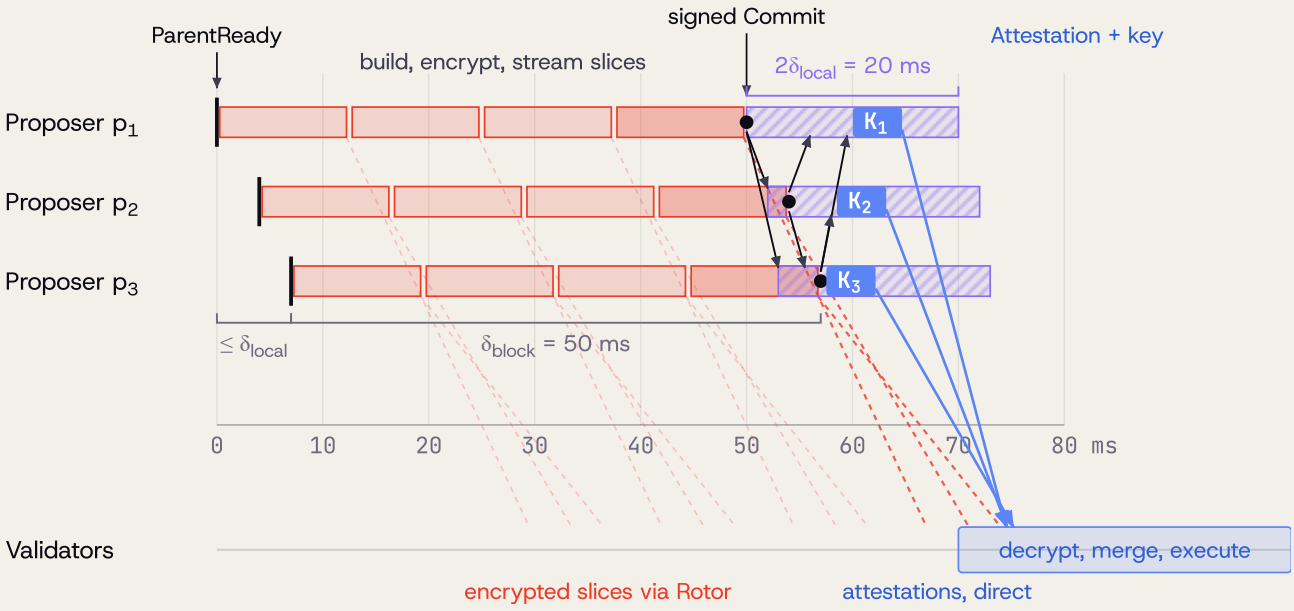



**Figure 3.1** One slot of a window with $k = 3$, drawn to scale on the proposers' side. Time in ms; each proposer builds and streams its block as encrypted slices for $\delta_{\text{block}}$ after its own ParentReady, sends its signed commitment at the slot's end, and waits inside its hatched $2\delta_{\text{local}}$ window, which opens at its first send or arrival. When every commitment is in, it releases its key (blue) in an attestation sent directly to all validators. The validator row is compressed: slices and attestations each cross the network once more before validators decrypt, merge and execute.

### 3.1.1 The proposer block

A proposer block has a header and an encrypted payload. The header names the slot $s$ and the proposer index $q$, and in the first slot of a window it also names a parent cblock. The payload holds the transactions. All slices of the

payload are encrypted under one fresh symmetric key $K_{s,q}$, one key per block, and the block is never sent at once: it is streamed out in slices throughout its slot.

In `acrux-node` the payload is a `BlockPayload`: the plain transactions the proposer holds for this slot or earlier, and the threshold fragments addressed to it (Chapter 5). `Network::begin_block` drains both from the proposer's mempool, encodes the payload, refuses one above 1 MiB, and cuts it into slices of 38,000 plaintext bytes. Each slice is encrypted separately under $K_{s,q}$. The header fields are carried in the metadata of every shred of the slice: slot, proposer, slice index, a flag on the last slice, the key hash $H(K_{s,q})$, the ciphertext length, and the parent reference, which is set only on the last slice of a block of the window's first slot.

### 3.1.2 The parent

Only the first cblock $c_{i_0}$ of a window chooses its parent. In its block of the window's first slot, each proposer names the cblock for which it emitted ParentReady, by slot and hash, normally the last cblock of the previous window. Co-proposers may name different parents; the parent of $c_{i_0}$ is the one named most often among the accepted blocks of that slot, ties broken by the lower proposer index. Every later cblock $c_i$ of the window has $c_{i-1}$ as its parent: its proposer blocks name none, and $h(c_i)$ includes $h(c_{i-1})$, so the four cblocks of a window form a hash chain.

### 3.1.3 Commitments

The commitment of a proposer block is its hash as disseminated, with the payload encrypted: $h_{s,q} = h(b_{s,q})$. It must also bind the key, so that $p_q$ cannot later reveal a different key that opens the same ciphertext to a different payload. Proposer $p_q$ sends its commitment to its co-proposers in the signed message `Commit(s, q, h_{s,q})`.

Acrux computes the commitment over a `BlockCommitment`: the slot, the proposer index, the key hash $H(K_{s,q})$, the parent reference and the Merkle root of every slice in slice order. Its SHA-256 is $h_{s,q}$. The roots bind every byte of ciphertext, since each root is computed over the slice's 64 coded shreds; the key hash binds the key. The `Commitment` message carries the slot, the proposer, $h_{s,q}$ and the key hash, and the proposer's Ed25519 signature covers its encoding.

### 3.1.4 Attestations

The attestation of proposer $p_q$ for slot $s$ is the signed message `Attestation(s, q, π, K_{s,q})`. The vector $\pi$ has one entry per proposer: $\pi[j] = h_{s,j}$ if $p_q$ received the commitment of $p_j$ on time, and $\pi[j] = \perp$ otherwise; a proposer always counts itself on time. An attestation thus states which proposers were on time and to which blocks they committed, and it reveals the key of the attester's own block. $P(\pi) = \{j : \pi[j] \neq \perp\}$ is the set of proposers attested by $\pi$, and two attestations *agree* if they carry the same

vector. A proposer *equivocates* if it signs two different proposer blocks, commitments or attestations for the same slot. Each of these is signed, so two of them are a proof, and equivocation is slashable.

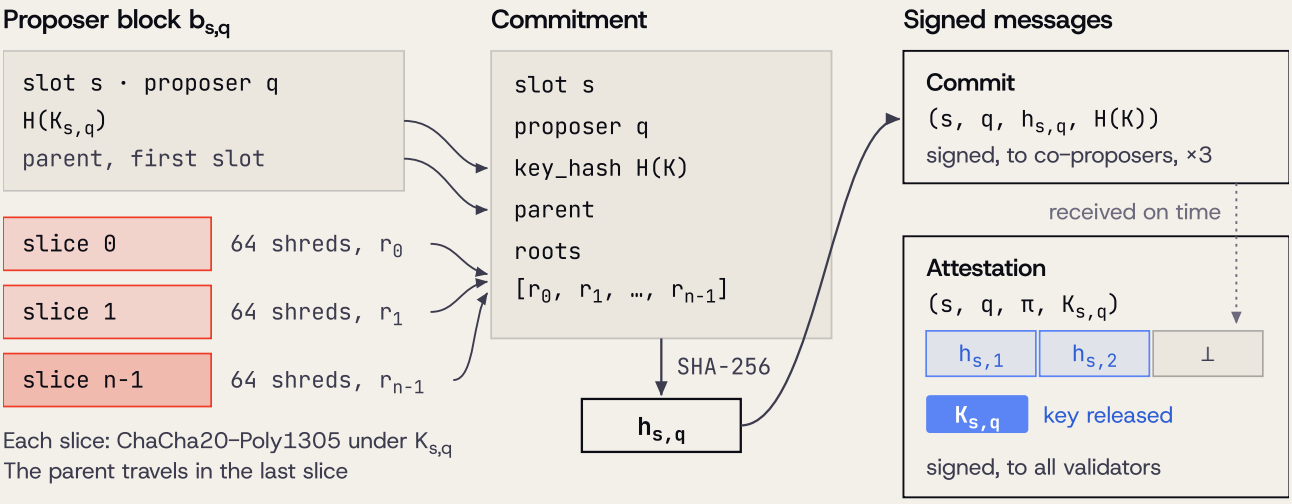



**Figure 3.2** What binds what. The commitment $h_{s,q}$ is the SHA-256 of the slice roots, the key hash and the parent, so it fixes every ciphertext byte and the key before either is revealed. The attestation carries the commitments its sender received on time, one entry per proposer, and the sender's own key. Byte layouts are in Appendix D.

**Table 3.1** The messages of a slot, who sends them and how they travel.

| Message | Fields | From → to | Path | Per slot |
|---|---|---|---|---|
| Shred | slice metadata, index, shard, root, Merkle proof, signature | proposer → all validators | Rotor: proposer to a relay, relay to all | 64 per slice |
| Commit | $s$, $q$, $h_{s,q}$, $H(K_{s,q})$ | proposer → co-proposers | direct, three copies | $k - 1 = 2$ per proposer |
| Attestation | $s$, $q$, $\pi$, $K_{s,q}$ | proposer → all validators | direct | one per proposer |
| Handover | the last cblock's shreds and attestations | proposers → next window's proposers | direct | last cblock of a window |
| Equivocation | two signed commitments of one proposer for one slot | co-proposer → all validators | direct | on evidence |

## 3.2 The proposer, Algorithm 1

Algorithm 1 is the paper's proposer, set here with its line numbers. Under each group of lines is the code in `backend/node/crates/acrux-node/src/` that carries it out, as `backend/spec/map.json` records it. $P$ is the set of proposer indices of the window, $p$ the running proposer, $i_0$ the window's first consensus slot, and consensus slot $i$ covers slots $i\kappa$, …, $i\kappa + \kappa - 1$. The notation is the book's (Chapter 2): the paper writes $\Delta$ for the delays, $\nu$ for $\kappa$ and $\tau$ for $\pi$.

**Algorithm 1** **Proposer $p$ of the window starting with consensus slot $i_0$. ParentReady is the event of its own Alpenglow instance and validator.**

```
P                                                          ▷ proposers for consensus slot i_o
start[s] ← ∞, done[s] ← false                                                ▷ for all slots s
received[s][j] ← ⊥, proposal[s][j] ← ⊥                  ▷ for all slots s and proposers j ∈ P
upon the first of ParentReady(i_o, ·) or its validator executing slot κi_o - 1 do
  for i = i_o, …, i_o + w/κ - 1 do                ▷ back to back, without waiting for votes/certs
    for s = iκ, …, iκ + κ - 1 do                                  ▷ back to back, without voting
```

`Network::start_window`, `Network::tick` in `protocol.rs`: a window opens on ParentReady; the tick starts slot $s$ once 50 ms × its index have passed since the window's start.

```
      K_s ← fresh symmetric key
      b ← pack transactions, encrypt payload with K_s
      stream the slices of b through Rotor during the next δ_block
```

`Network::begin_block`: `crypto::fresh_key`, 38,000-byte slices, `encrypt_slice`, `encode_slice`; a task sends slice $j$ of $n$ at $(j + 1)/n$ of the slot, less 1 ms.

```
      if s = i_oκ then                                                ▷ first slot of the window
        wait for ParentReady(i_o, h(c_j)), keep streaming until δ_block after it
        b.parent ← (j, h(c_j))                                  ▷ parent c_j, in the last slice
      end if
```

`Network::prepare_early` streams the prefix slices once the previous window is executed; `Network::complete_early` adds the last slice with the parent, 49 ms after ParentReady.

```
      start[s] ← min(now, start[s])
      proposal[s][p] ← h(b)
      send signed Commit(s, p, h(b)) to all co-proposers, a few times                ▷ redundancy
```

`ExchangeEngine::launch` in `exchange.rs`: wakes at the slot's end, records its own send, sends each commitment three times, 250 µs apart.

```
      if s is the last slot of the window then send b directly to the next proposers
```

`Network::try_vote` sends `Wire::Handover` with the last cblock; `Network::receive` rebuilds it at the next proposers.

```
    end for
  end for
```

`Network::tick`: sixteen slots per window, then the next ParentReady.

```
upon receiving first Commit(s, q, h) from a co-proposer q do
  start[s] ← min(now, start[s])
  received[s][q] ← now
  proposal[s][q] ← h
```

`ExchangeEngine::receive_commit`: called right after the signature check, before any validator queue; a second, different commitment becomes `Wire::Equivocation`.

```
upon ¬done[s] and proposal[s][p] ≠ ⊥ and (received[s][q] ≠ ⊥ for all q ≠ p or now > start[s]
+ 2δ_local) do
  done[s] ← true
  π = [⊥, …, ⊥]
  for q ∈ P do
    if q = p or received[s][q] ≤ start[s] + 2δ_local then          ▷ received[s][q] = ⊥ → late
      π[q] ← proposal[s][q]
    end if
  end for
  send signed Attestation(s, p, π, K_s) to all validators
```

`ExchangeEngine::launch` waits on a notify or the deadline; `Exchange::attestation_vector` builds $\pi$ and returns nothing until the own commitment is sent.

### 3.2.1 Window start

A proposer starts its window when its own Alpenglow instance emits ParentReady for $i_0$, or earlier, when its validator has executed the last slot of the previous window (line 4, and Section 3.5). Its co-proposers emit ParentReady at about the same time: they are co-located and receive the same certificates, and every proposer forwards the certificates that triggered its ParentReady directly to its co-proposers, so the events differ by at most $\delta_{\text{local}}$. In `acrux-node`, a proposer that receives a certificate forwards it, with the signed votes it was formed from, to the proposers of that window and of the next; the Votor of each node turns the certificates into ParentReady (Section 4.8), and `Network::start_window` records the window's start on that node's own clock.

From then on the proposer produces all $w$ blocks of its window back to back, one every $\delta_{\text{block}}$, and waits for no vote or certificate (lines 5 and 6). `Network::tick` starts slot $s$ when $s$ mod 16 times 50 ms have passed since the window's start, so a node's slot boundaries come from one local clock and do not drift against each other within a window. On Acrux Net, slot pacing within a window runs at p50 50.0 ms and p99 51.3 ms (Chapter 9).

### 3.2.2 Building and sealing a block

In every slot the proposer draws a fresh 32-byte key $K_s$ from the operating system's random source (line 7), packs the transactions users sent to it, and encrypts the payload (line 8). The key is new for every block: a key released for slot $s$ opens no other block, of this proposer or any other.

Acrux encrypts each slice with ChaCha20-Poly1305 under $K_{s,q}$. The 12-byte nonce is derived from the slot, the proposer index and the slice index, and the same three values are the associated data, so a slice decrypts only in its own position: moved to another slot, proposer or index, it fails authentication. A slice carries at most 65,520 plaintext bytes, 65,536 with the tag. The encryption is `encrypt_slice` in `backend/node/crates/acrux-crypto/src/lib.rs`; its inverse, `decrypt_slice`, is the one validators run once the key arrives (Section 4.5).

### 3.2.3 Why the commitment binds the key

The commitment covers the ciphertext. If it covered nothing else, a proposer could look for a second key under which the same ciphertext decrypts to a different payload and release that key instead, choosing its block after it has seen the others. The paper therefore requires the commitment to bind $K_{s,q}$, for example by including $H(K_{s,q})$ in the header [1]. Acrux does so twice over: $H(K_{s,q})$ is a field of the `BlockCommitment` that $h_{s,q}$ hashes, and it is a field of every shred's signed metadata. A validator checks $H(K)$ of the released key against that field before it decrypts a slice, and refuses the block if they differ. With a key fixed by its hash and an authenticated cipher, a released key opens exactly the committed block or nothing. The argument is Lemma 2.3, proved in Chapter 7.

## 3.3 Dissemination through Rotor

Every slice is erasure coded into $\Gamma = 64$ shreds, any $\gamma = 32$ of which reconstruct it, as in Alpenglow [3]. With shreds of about 1.2 KB a slice carries about 38 KB, small enough to send at least one slice in every 50 ms slot. Rotor relays are sampled separately for each slot, each slice index and each co-proposer, so the relays of the three blocks of a slot are independent draws [1].

`encode_slice` cuts the ciphertext of a slice into 32 data shards of equal length, the last padded with zeros, and adds 32 coding shards with Reed-Solomon over GF($2^8$). The 64 shards are the leaves of a Merkle tree of depth 6. Leaves and inner nodes are hashed under different one-byte prefixes and each leaf includes its index, so a leaf can never pass for an inner node or for another position. Every shred carries its shard, the slice's root, its six-hash proof and the proposer's signature over the shred tag, the slice metadata, its index, the root and the hash of its shard. The signature binds the metadata as well as the root: the key hash, the last-slice flag, the parent and the ciphertext length cannot be changed in transit.

Each of the 64 shreds goes to one relay, drawn by stake from a seeded mix of the slot, the slice index, the proposer index and the shred index (`LocalTransport::relay` in `backend/node/crates/acrux-transport/src/transport.rs`). The proposer sends each relay its shreds in one `Wire::Shreds` packet; the relay checks that the sender is the scheduled proposer and forwards them to every validator as `Wire::Relay`. Every node verifies the signature, the proof against the root, the proof depth and the shard length of each shred as the packet arrives, before it reaches the validator (`authenticate` in `protocol.rs`). When 32 distinct shreds of a slice are in, it rebuilds the ciphertext (`reconstruct_verified_slice`). Two shreds with the same position but different content, or a shred whose root or metadata differs from the slice's first, mark the proposer as having equivocated for the slot, and both shreds are stored as evidence.

Slices leave throughout the slot rather than at its end. Of a block of $n$ slices, slice $j$ is sent at $(j + 1)/n$ of $\delta_{\text{block}}$ after the slot's start, less 1 ms, so the last slice goes out 49 ms into the slot, one millisecond before the commitment.

## 3.4 The hash exchange

Together with the last slice of $b_s$, the proposer sends `Commit(s, p, h(b_s))` directly to its co-proposers (line 16). It times the exchange from $start[s]$, measured on its own clock: the earlier of the moment it sends its own commitment (line 14) and the moment it receives the first commitment of a co-proposer for slot $s$ (line 21). A co-proposer is **on time** if its commitment arrives by $start[s] + 2\delta_{\text{local}}$, and late otherwise.

$2\delta_{\text{local}}$ = 20 ms on Acrux Net; messages inside a hub take 1 to 4 ms.

The interval is $2\delta_{\text{local}}$ because co-proposers do not start a slot at exactly the same moment. Their ParentReady events differ by up to $\delta_{\text{local}}$, so a correct co-proposer may send its commitment up to $\delta_{\text{local}}$ later, and the message itself takes up to another $\delta_{\text{local}}$. Since *start*[*s*] is never later than its own sending time, a proposer waits at most $2\delta_{\text{local}}$ after sending its own commitment. Anchoring the window at the first arrival rather than at the proposer's own send matters: a proposer that sends late cannot stretch the window it grants the others. No clocks are synchronized; the window opens on a local event and is timed by the local clock [1].

Acrux runs the exchange in its own engine, `ExchangeEngine`, on a timer runtime separate from execution, so a busy validator never holds back a commitment or an attestation. `launch` sleeps until the slot's end, 50 ms after its start, records its own send, sets *start*[*s*] to the earlier of that send and any commitment already received, and sends the commitment. `receive_commit` is called on a commitment as soon as its signature is checked, before the packet enters the validator's queue, so the arrival time it records is the time the packet arrived. The deadline is inclusive: in the engine's tests a commitment arriving at exactly *start*[*s*] + 20 ms is on time, and one at 20.001 ms is late.

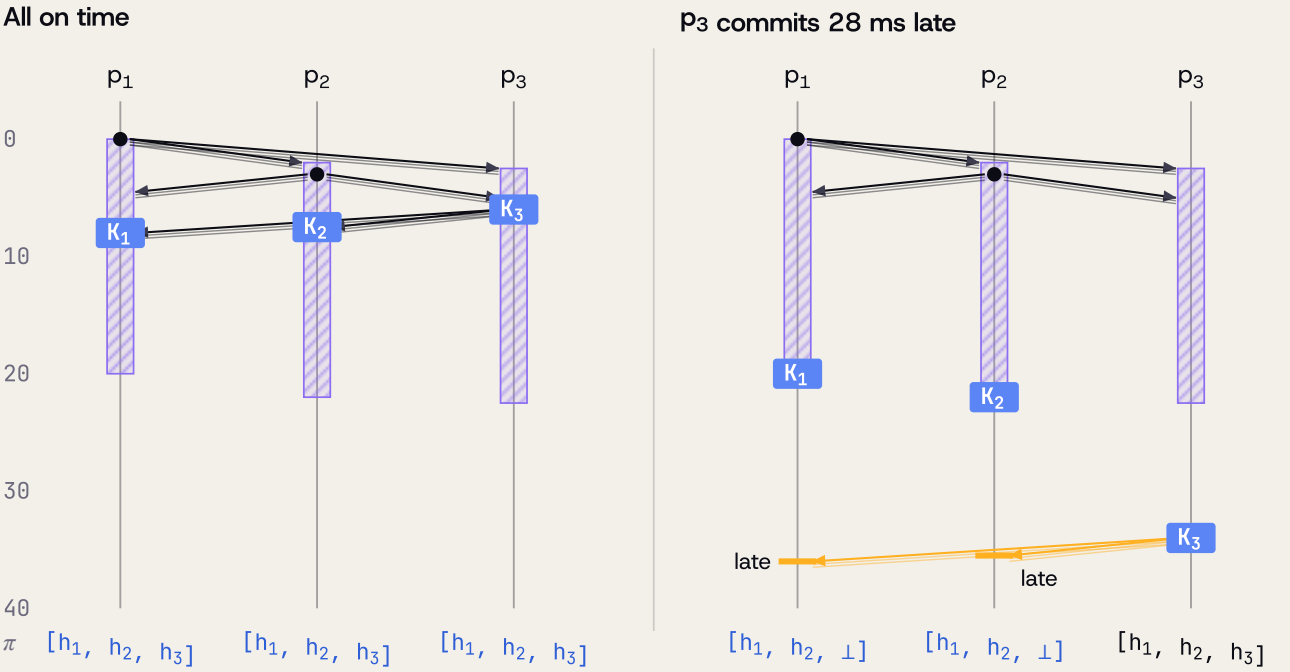



**Figure 3.3**

The hash exchange of one slot, time downward in ms from the first commitment, each commitment sent three times. Left: all three commitments land inside every window, and the three attestations carry the same vector. Right: $p_3$ commits 28 ms late, as in the Fault Lab's last-look scenario. Its commitment reaches $p_1$ and $p_2$ after their windows close (amber), so they attest it as $\perp$ at their deadlines, while $p_3$ received theirs on time and attests all three. The vectors disagree, and validators vote skip.

### 3.4.1 The robust hash exchange

Commitments are the only messages of the protocol without built-in redundancy. Slices are erasure coded and relayed by Rotor, attestations and votes go to every validator, so a few lost packets do not matter for them. A

commitment is one message from one proposer to one co-proposer: if it is lost, the receiver marks an on-time sender late, the attestations disagree and the cblock may be skipped. Commitments are few, though, $k - 1 = 2$ per proposer and slot, so they are cheap to protect. The paper's simple measure is to send every commitment several times, a fraction of a millisecond apart, since any copy that arrives counts [1].

Acrux sends every commitment three times to each co-proposer, at 0, 250 and 500 µs after its send. The receiver keeps the first copy; a later copy with the same hash is ignored, and a copy with a different hash is equivocation. The paper also gives an optional second round, in which every proposer forwards each commitment it receives to its other co-proposers, which widens the waiting interval to $3\delta_{\text{local}}$. Acrux Net runs without it, on the $2\delta_{\text{local}}$ interval. The node also runs at high process and thread priority, so the host's scheduler does not delay a commitment past $2\delta_{\text{local}}$.

### 3.4.2 The attestation vector

As soon as the commitments of all co-proposers have arrived, and at the latest at $start[s] + 2\delta_{\text{local}}$, the proposer builds $\pi$ (lines 24 to 31): its own commitment in its own entry, the commitment of every co-proposer that arrived on time in that co-proposer's entry, $\perp$ for every other. It then sends `Attestation(s, p, π, K_s)` directly to all validators (line 32). It never attests before it has sent its own commitment, even when every co-proposer's commitment arrived earlier: `Exchange::attestation_vector` returns nothing until the own send is recorded, and an early co-proposer commitment cannot displace the proposer's own entry. Only now, in the attestation, does the key leave the proposer.

An entry carries the hash itself, not a bit. A validator that later rebuilds $b_{s,q}$ from its shreds recomputes the commitment from the slice roots, the key hash and the parent, and compares it with $\pi[q]$ of the attestations it holds (validity condition (iv), Section 4.3). So an attestation fixes not only who was on time but which block of theirs it accepts.

| 4.7 ms | 12.0 ms | 48.5 ms | 76.1 ms |
|---|---|---|---|
| Commit spread p50, first to last commitment of a slot | Commit spread p99, inside the 20 ms window | Attestation arrival at validators p50, after the first commitment | Attestation arrival p99, validators in every hub |

The readings above are from 600 s of Acrux Net without faults (Chapter 9). Attestations reach validators in all three hubs at about the time the blocks do: an attestation waits for the local exchange and then goes straight to every validator, while a block takes Rotor's extra relay hop. As long as $2\delta_{\text{local}} \le \delta_{\text{global}}$, 20 ms against 250 ms here, the exchange adds nothing to what a validator waits for.

### 3.4.3 Proposal hiding

> **When a proposer releases its key it has already committed to its own block, and its block can be accepted only together with the blocks whose commitments it attests, all fixed before any payload was revealed.**

If a block reached the co-proposers in the clear, the slowest proposer could read every other block and change its own: the last look. The encryption and the tie between key release and attestation close it. Until a proposer receives attestations it holds at most the others' ciphertext. A co-proposer that holds back its commitment to see more either commits before anything is revealed, or misses the $2\delta_{\text{local}}$ window, is attested late and loses its block. The cost is a few bytes in the attestation and the execution delay of opening no block before every proposer has committed [1]. The Fault Lab's *last look* scenario runs exactly this attack, a proposer committing 28 ms late, and records the dropped block and the skip certificate (Chapter 10).

---

## 3.5 Fast handover to the next window

In Alpenglow the leader sends its last block directly to the next leader. In New Constellation the proposers send the full cblock of the window's last consensus slot directly to the proposers of the next window (line 17), so these learn the latest transactions sooner and do not include them again. They usually hold the last cblock $c_{i_0-1}$ before its certificate arrives, and they do not wait for the certificate: a proposer starts to build and stream its first block as soon as its own validator has executed $c_{i_0-1}$, or when it emits ParentReady, whichever comes first [1].

The early start never forces a proposer to switch parents. Only the first block of a window depends on the parent, and the parent reference, though part of the header, travels in its last slice (lines 10 to 12). The proposer still completes that block $\delta_{\text{block}}$ after its ParentReady, exactly as without the early start, so co-proposers complete their first blocks at most $\delta_{\text{local}}$ apart however early each began. Validators lose nothing, since they cannot decrypt the block before its attestation anyway.

In `acrux-node` the handover has three parts:

- **The last cblock.** When a proposer assembles and votes for the window's fourth cblock, `Network::try_vote` sends `Wire::Handover` to the next window's proposers: the slot, the cblock reference, and every shred and attestation of its four slots. The receivers check each attestation's signature, rebuild the slices and execute the cblock without waiting for Rotor.

— **The early prefix.** As soon as a node of the next window's proposers has executed the window's last slot, `Network::prepare_early` draws the key of its first block, packs its mempool, and encrypts and streams every slice of the payload, none of them marked last and none carrying a parent.

— **The last slice.** On ParentReady, `Network::complete_early` encrypts one more, empty slice, marked last and carrying the parent reference, computes the commitment over all slice roots, sends that slice 49 ms after ParentReady and hands the commitment to the exchange engine, which sends it at 50 ms.

At the same boundary, each proposer of the window before hands the transactions and fragments it holds for earlier slots but did not include to the proposer in the same role of the new window, which accepts them into its mempool. A transaction sent for a slot of a skipped window is carried by the next window's proposers. The time from a window's last slot to the next window's first, per pair of hubs, is measured in Chapter 9.

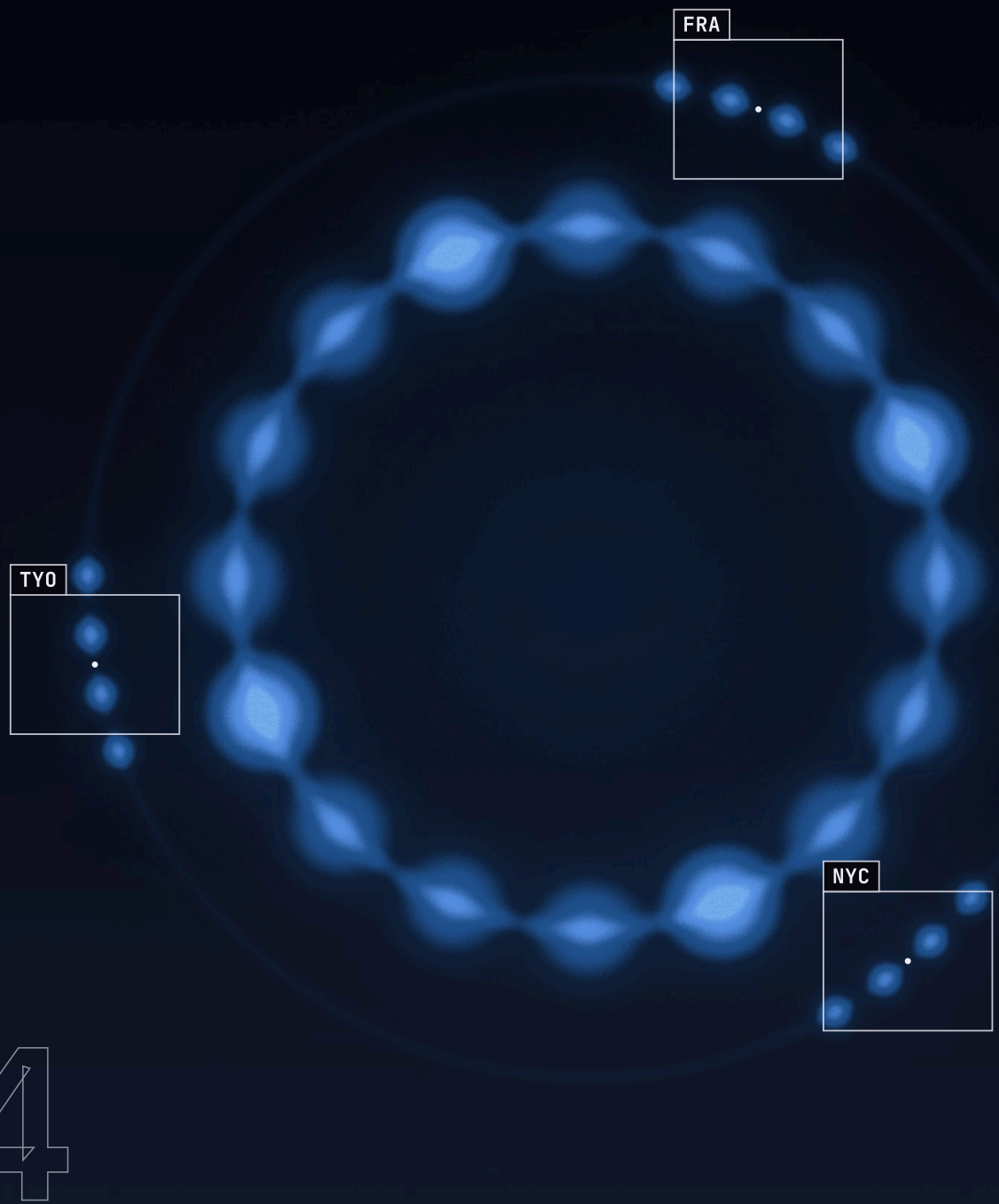
## 3.6 A crashed proposer

A proposer that is crashed from the start of its window sends nothing: no slices, no commitment, no attestation. Its co-proposers run the exchange as always. Each sets $start[s]$ at its own send or at the first co-proposer commitment, waits until $start[s] + 2\delta_{local}$, finds no commitment from the crashed proposer and writes $\perp$ in its entry. With $k = 3$ and proposer 3 down, both attestations carry $\pi = [h_{s,1}, h_{s,2}, \perp]$. They agree, and the attested set $P = \{1, 2\}$ is a strict majority, so validators accept the slot on the two blocks (Section 4.4). The window costs each co-proposer $2\delta_{local}$ of waiting per slot and nothing else.

A late proposer is a different case from a crashed one. A proposer that is alive but commits after its co-proposers' windows close is attested late by them, while its own attestation names all three commitments, since it received theirs on time. The vectors disagree, no set is agreed, and validators vote skip for the cblock: the late block is never executed, and no subset of the attested blocks is ever accepted in its place. Figure 3.3 shows both vectors.

`acrux net up --crash-window W --crash-proposer Q` keeps one proposer silent for a whole window: `Network::tick` skips its `begin_block`, and `prepare_early` skips its early start. In a 120 s run of Acrux Net with proposer 3 of window 5 silent, 526 cblocks were finalized and none skipped, and the window ran on $P = \{1, 2\}$. The Fault Lab's *crash* scenario records the same answer as evidence: slot 64 agreed on $P = \{1, 2\}$ and the window continued on two proposers (Chapter 10). What validators do while the third attestation is missing, and why only the window's first vote waits, is in Section 4.6.1.

NOTARIZE 60% · FAST FINALIZE 80%
$\Delta_{timeout}$ 750 MS = $3\delta_{global}$
ONE VOTE PER CBLOCK OF 4 SLOTS




4

# Validating and voting

A validator accepts a slot only when every attestation it holds agrees on the set P. Four slots make a consensus block, and Alpenglow votes on its state.

Every validator, proposer or not, rebuilds the blocks of each slot from Rotor, collects the attestations, and accepts the slot only when all of them agree on one set of proposers. It decrypts and executes those blocks in order, and votes once per cblock of four slots on the state they produce.

## 4.1 What a validator does with a slot

A validator receives the shreds of every proposer block through Rotor and the attestations directly from the proposers (Chapter 3). From these it decides, slot by slot, which proposers' blocks it accepts, executes them, and assembles the cblock that Alpenglow votes on. The paper specifies this as Algorithm 2 [1]. Alpenglow itself runs unchanged on consensus slots and cblocks; only its timeouts and its Block events come from this algorithm [3].

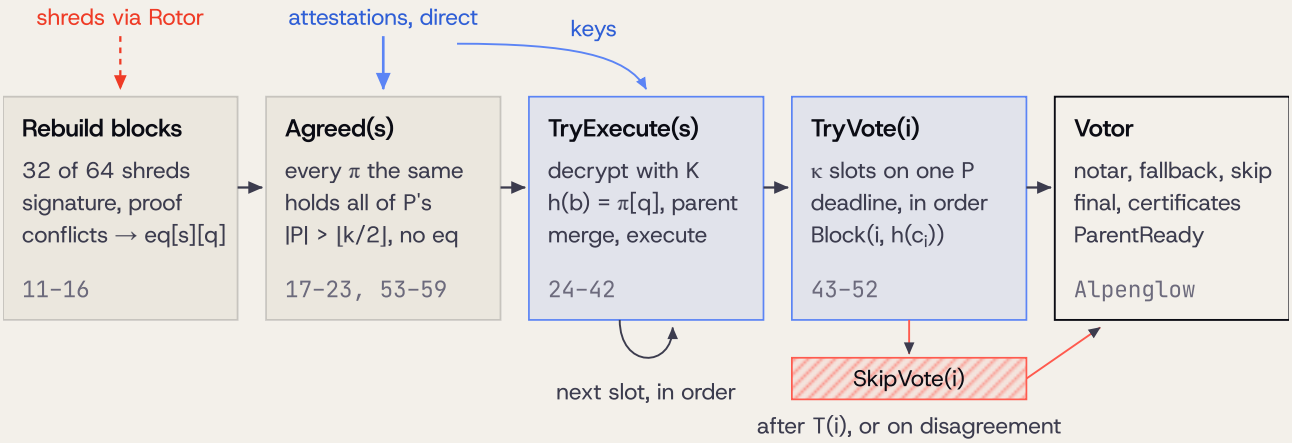



Figure 4.1

The validator's path through one slot and one cblock, with the lines of Algorithm 2 that carry each step. Blocks and attestations arrive independently; a slot executes once it is agreed and its blocks are in, slots execute in order, and every fourth slot completes a cblock that the validator votes on. Blue steps exist only in New Constellation; Votor is Alpenglow's.

## 4.2 The validator, Algorithm 2

Algorithm 2 is set here with the paper's line numbers. $P[s]$ is the agreed set a validator executed slot $s$ with, $x[s]$ the xblock, $\sigma[s]$ the state after slot $s$, $b[s][q]$ the block of proposer $q$, $V[s][q]$ its attestation, and eq[s][q] whether it equivocated. The book's notation follows Chapter 2: where the paper writes $\Delta$ for the delays, $\nu$ for $\kappa$, $\tau$ for the vector $\pi$, $\Sigma$ for the states and $\pi$ for the chosen parent of line 32, this listing writes $\delta$, $\kappa$, $\pi$, $\sigma$ and $\hat{c}$; $\sigma'$ is the state a slot builds on. Under each group of lines is the code that carries it out, in `backend/node/crates/acrux-node/src/protocol.rs` unless named otherwise, as `backend/spec/map.json` records it.

**Algorithm 2** **Validator. Alpenglow runs unchanged on consensus slots and cblocks, except that its timeouts and its Block events come from this algorithm.**

```
1  P[s], x[s], σ[s] ← ⊥                                                  ▷ for all slots s
2  b[s][q], V[s][q] ← ⊥; eq[s][q] ← false                  ▷ for all slots s and proposers q
3  voted[i] ← ⊥                                                  ▷ for all consensus slots i
4  upon ParentReady(i_o, ·) do
5    for i = i_o, …, i_o + w/κ - 1 do
6      Timeout(i) ← now + Δ_timeout + (i - i_o + 1)κδ_block + δ_block
7      Deadline(i_o) ← now + δ_block + 2δ_global + 2δ_local
8    end for
```

`LocalSlot` and `WindowRun` hold the per-slot and per-window state; `Network::start_window` records ParentReady's time; `Network::try_vote` measures every timeout and the deadline from it.

```
9  upon now = Deadline(i_o) do                          ▷ stop waiting for missing attestations
10   TryVote(i_o)
```

`Network::tick` calls `Network::try_vote` for the live windows on every tick, so the vote leaves as soon as the deadline has passed.

```
11 upon receive new shred of proposer q for slot s from Rotor do
12   if saw equivocation evidence then eq[s][q] ← true
13   if reconstructed a valid block b_{s,q} of q for slot s then
14     b[s][q] ← b_{s,q}
15     TryExecute(s)
16   end if
```

`Network::receive`, `Wire::Relay`: piles shreds per proposer and slice, marks conflicting shreds as equivocation, rebuilds a slice at 32.

```
17 upon V_{s,q} = Attestation(s, q, π, K) do
18   if V[s][q] ≠ ⊥ then
19     if V_{s,q} ≠ V[s][q] then eq[s][q] ← true                  ▷ equivocation, slashable
20   else
21     V[s][q] ← V_{s,q}
22     TryExecute(s); TryVote(⌊s/κ⌋)
23   end if
```

`Network::receive`, `Wire::Attest`: checks the attester is proposer *q* of the slot's window, keeps the first, stores a different second one as evidence.

```
24 procedure TryExecute(s)
25   i ← ⌊s/κ⌋; A ← Agreed(s)
26   if P[s] ≠ ⊥ or A = ⊥ or b[s][q] = ⊥ for some q ∈ A then return
27   if s is not the first slot of its window and P[s - 1] ≠ A then return
28   x[s] ← (b_{s,q})_{q∈A}
29   T_q ← Dec_{V[s][q].K}(b[s][q]) for all q ∈ A
30   if some T_q is invalid or h(b[s][q]) ≠ V[s][q].π[q] for some q ∈ A then return
```

`Network::try_execute`: walks the window's slots in order, stops at the first not agreed, incomplete or invalid; recomputes each commitment and checks the key hash before decrypting.

```
31   if s is the first slot of its window then
32     ĉ ← the most common b[s][q].parent over q ∈ A, resolve ties to the lowest q
33     if σ[ĉ] = ⊥ then return                                  ▷ parent not executed yet
34     σ′ ← σ[ĉ]; par[i] ← ĉ
35   else
36     σ′ ← σ[s - 1]; if s = iκ then par[i] ← (i - 1, h(c_{i-1}))
37   end if
```

`Network::try_execute`: in a window's first slot, restores the executor and the fragment cache from the chosen parent's

checkpoint, or waits for the parent.

```
38   σ[s] ← Execute(σ', Merge(s, (T_q)_{q∈A}))                    ▷ e.g. Algorithm 4
39   P[s] ← A
40   if s = iκ + κ - 1 then TryVote(i)
41   TryExecute(s + 1)                                  ▷ slot s + 1 may be complete already
42 end procedure
```

`SvmExecutor::execute_slot` in `acrux-execution/src/execution.rs`: merges, selects on four tracks, executes on the SVM, returns the state hash.

```
43 procedure TryVote(i)                                  ▷ emits Block for c_i, at most once
44   if voted[i] ≠ ⊥ or P[s] = ⊥ or Agreed(s) ≠ P[s] for a slot s of c_i then return
45   first ← i is the first consensus slot of its window
46   if first and V[iκ][q] = ⊥ for some q and now < Deadline(i) then return          ▷ wait
47   if not first and voted[i - 1] = ⊥ then return                  ▷ vote in order
48   voted[i] ← P[iκ]
49   c_i = (par[i], (x_s)_{s∈{iκ,…,iκ+κ-1}}, h(σ[iκ + κ - 1]))          ▷ with the state hash
50   emit Block(i, h(c_i), par[i])
51   TryVote(i + 1)                                  ▷ c_{i+1} may be waiting for this vote
52 end procedure
```

`Network::try_vote`: hashes the cblock, hands it to `Votor::local_block`, or calls `Votor::timeout` for a skip vote.

```
53 function Agreed(s)                          ▷ the accepted proposers of slot s, or ⊥
54   if V[s][q] = ⊥ for all q, or eq[s][q] for some q then return ⊥
55   if V[s][q].π ≠ V[s][q'].π for some V[s][q], V[s][q'] ≠ ⊥ then return ⊥   ▷ all agree
56   A ← {p : π[p] ≠ ⊥} with π = V[s][q].π for some V[s][q] ≠ ⊥
57   if |A| ≤ ⌊k/2⌋ or V[s][q] = ⊥ for some q ∈ A then return ⊥
58   return A
59 end function
```

`agreed` in `agreement.rs`, called by `Network::try_execute`; `Network::try_vote` re-checks agreement over the cblock's slots.

## 4.3 Validity of a proposer block

A proposer block $b_{s,q}$ is valid if it meets seven conditions [1]. A validator does not check them in one place: each is checked where its evidence arrives, some on the network path before a packet reaches the validator, the rest when the slot is about to execute. Table 4.1 sets each condition beside what it guards and the code that checks it.

**Table 4.1** The validity conditions (i) to (vii) of the paper, their purpose, and where acrux-node checks them.

| | Condition | What it guards | In acrux-node |
|---|---|---|---|
| (i) | All shreds are valid and signed by $p_q$, their Merkle roots match, and with the reconstructed shreds they reproduce the root in the received shreds | Every byte a validator rebuilds is the byte the proposer coded and signed | `verify_shred` at ingress: signature, proof of depth 6, shard length; every shred of a slice must carry the same root and metadata; `reconstruct_verified_slice` at 32 |
| (ii) | The block is complete and was received before the timeout of its consensus slot | A vote never waits on a block that may not come | `try_execute`: exactly one last slice and every slice before it rebuilt; `try_vote`: no cblock vote after $T(i)$ |

| | Condition | What it guards | In acrux-node |
|---|---|---|---|
| (iii) | It is the only block of $p_q$ for slot $s$ | One proposer, one block per slot | Conflicting shreds, commitments or attestations set the slot's equivocation flag and are stored as evidence |
| (iv) | The block hashes match the ones the attestations attest to | The executed block is the committed one | `try_execute` recomputes $h($ `BlockCommitment` $)$ from the slice roots, key hash and parent, and compares it with $\pi[q]$ |
| (v) | Its payload decrypts under $K_{s,q}$ of $p_q$'s attestation, and the key matches the commitment | No second key opens the same ciphertext | $H(K)$ against the key hash, then ChaCha20-Poly1305 authentication of every slice |
| (vi) | In the first slot of a window, its parent reference is correct | The window builds on a cblock the validator has executed | The parent is in the last slice and inside the commitment; `try_execute` waits until the chosen parent is executed |
| (vii) | It respects size, format and compute limits | Execution is bounded and well formed | Payload decodes as `BlockPayload`; each transaction re-validated against its signed header with target slot $\leq s$; fragments re-validated and addressed to $q$; fee-payer inclusion within $\rho/(2k)$ per window; slices at most 64 KiB; 2,000,000 CU per track |

Conditions (i) and (iii) are settled as shreds arrive. Every node verifies each shred's signature and Merkle proof in `authenticate`, before the packet reaches its validator, so the pile a validator rebuilds from holds only authenticated shreds. Two shreds for the same position with different content, or a shred whose root or metadata differs from the slice's first, set the slot's equivocation flag (line 12). Two different commitments from one proposer, seen by a co-proposer, become a `Wire::Equivocation` message to all validators that carries both signed commitments; two different attestations from one proposer for one slot do the same at every validator (line 19). In each case the two signed messages are the slashing proof, and the slot is no longer agreed.

Conditions (iv) and (v) bind the executed block to the attested one. The validator recomputes the commitment of each block in the agreed set from what it rebuilt: the slice roots in order, the key hash from the shred metadata, and the parent from the last slice. It requires the result to equal $\pi[q]$ (line 30). Before decrypting a slice it requires $H(K)$ of the released key to equal the key hash in the metadata; a slice that fails ChaCha20-Poly1305 authentication makes the block invalid. Condition (vii) includes the fee-payer rate of Chapter 5: a block that includes transactions of one fee payer with inclusion fees above $\rho/(2k)$ in a window is invalid, which on Acrux Net is 166,666 lamports per fee payer, proposer and window.

## 4.4 Agreement

A validator considers slot $s$ **agreed** on the set $P_s$ if [1]:

| | |
|---|---|
| **(a) Unanimous** | all attestations for slot $s$ it has received agree, and $P_s$ is the set they attest; |
| **(b) Complete** | it holds the attestation of every proposer in $P_s$; |
| **(c) Majority** | $P_s$ is a strict majority of the proposers, $\lvert P_s\rvert > \lfloor k/2 \rfloor$, so at least 2 of 3; |
| **(d) Clean** | it has seen no equivocation by any proposer in slot $s$. |

The function `agreed` in `agreement.rs` is Agreed($s$) of lines 53 to 59. It returns nothing on equivocation or without attestations, requires every attestation to name slot $s$, come from proposer 1, 2 or 3 and carry a vector of length 3 equal to the first one's, takes $A$ as the indices whose entry is a hash, and returns $A$ only if it has more than one member and an attestation from each. Its unit test covers the cases one by one: three agreeing vectors give {1, 2, 3}; two attestations naming {1, 2} give {1, 2}; three vectors of all three with one attestation missing give nothing, since an attested member is still missing; one proposer attesting only itself gives nothing; a late mark that only one attester makes gives nothing.

Normally $P_s$ holds all $k$ proposers. A crashed proposer is attested late by all its co-proposers, the attestations still agree, and $P_s$ holds the others; with $k = 3$ a window tolerates one crashed proposer, while with $k = 2$ a crash would leave no majority. A slot can be agreed while the attestation of a proposer outside $P_s$ is still missing. If that attestation arrives and disagrees, the slot is no longer agreed, and the validator votes skip for its cblock.

Within a window the agreed set cannot change: a slot that is not the window's first executes only if it is agreed on the same set as the slot before it (line 27). A proposer dropped from $P$ in the first slot stays out for the window, and a proposer cannot be dropped later without stopping execution of the window.

## 4.5 Executing a slot

Once slot $s$ is agreed on $P_s$ and the validator holds valid blocks from every proposer in it, the validator decrypts these blocks with the keys from their attestations, merges their transactions into one deterministic order, and executes the resulting xblock $x_s$. Blocks of proposers outside $P_s$ are ignored. Slots execute in order, each on the state after the previous slot, and the first slot of a window on the state after its parent cblock [1].

### 4.5.1 TryExecute

`Network::try_execute` runs on every tick for the live windows of a node and walks the window's sixteen slots from the first not yet executed. It stops at the first slot that a skip certificate has cut off, that has no agreed set, whose set differs from the previous slot's, whose blocks are incomplete, or whose blocks fail a check. A slot that stops it is retried on the next tick, so arrival

order does not matter: whichever of the last shred and the last attestation comes second makes the slot executable, as lines 15 and 22 express by calling TryExecute from both events.

For each proposer of the agreed set, `try_execute` finds the one last slice, decrypts every slice up to it under the released key, rebuilds the commitment and compares it with $\pi[q]$, and decodes the payload. In the window's first slot it then chooses the parent: the parent named most often among the accepted blocks, ties to the lower proposer index (line 31). If that cblock is not yet executed here, it waits (line 32). Otherwise it restores the executor and the threshold-fragment cache from the checkpoint taken after the parent, so the window runs on exactly the parent's state.

### 4.5.2 Duplicates, merge and execution

The same transaction may arrive in the blocks of several proposers, since users may send to more than one proposer of a slot (Chapter 5). `SvmExecutor::execute_slot` merges the decrypted transactions by their ID, the hash of the signed message, and records which proposers carried each copy. A transaction already executed in an earlier slot is dropped. Each included copy pays its inclusion fee to the proposer that carried it, and the transaction executes once. Threshold fragments are added to the fragment cache, and a transaction whose $t$-th distinct fragment arrives in this slot is decoded and joins the merge.

The merged transactions are sorted by priority fee per compute unit, selected onto $m = 4$ execution tracks of 2,000,000 CU each by Algorithm 4, and executed in that order on the SVM through LiteSVM. Each transaction is re-validated against its signed bytes at this boundary, so a public header cannot redirect fees or writable accounts. The order, the selection and the fees are the subject of Chapter 5; for this chapter it matters that all of it is a deterministic function of the agreed blocks and the parent state, so every correct validator that executes slot $s$ on $P_s$ reaches the same state $\sigma[s]$.

### 4.5.3 The state hash

A lattice hash: changing one account costs one subtraction and one addition.

The state hash is an additive accumulator over all accounts. For each account the executor hashes, with SHA-256 under the domain tag `acrux-account-v1`, its address, lamports, owner, executable flag, rent epoch, data length and data. The state hash is the sum of these hashes as 256-bit integers modulo $2^{256}$. When an account changes, the executor subtracts its old hash and adds its new one, so the state hash after a slot costs work proportional to the accounts the slot touched, not to the state. After the last slot of every consensus slot the executor takes a checkpoint, which a later window's first slot restores when it builds on that cblock. Across the twelve validators of Acrux Net, every executed slot's state hash is compared; in the 600 s fault-free run, 10,564 comparisons found no mismatch (Chapter 9).

## 4.6 Voting on a cblock

A validator casts a notarization vote `NotarVote(i, h(c_i))` for consensus slot $i$ if, before its timeout $T(i)$, all $\kappa$ slots of $c_i$ are agreed on the same set $P$ and it has validated and executed the blocks of every proposer in $P$. Otherwise it casts `SkipVote(i)`. In particular it votes skip when two attestations disagree, for example because one proposer considers another late. It never votes for a subset of the attested blocks [1].

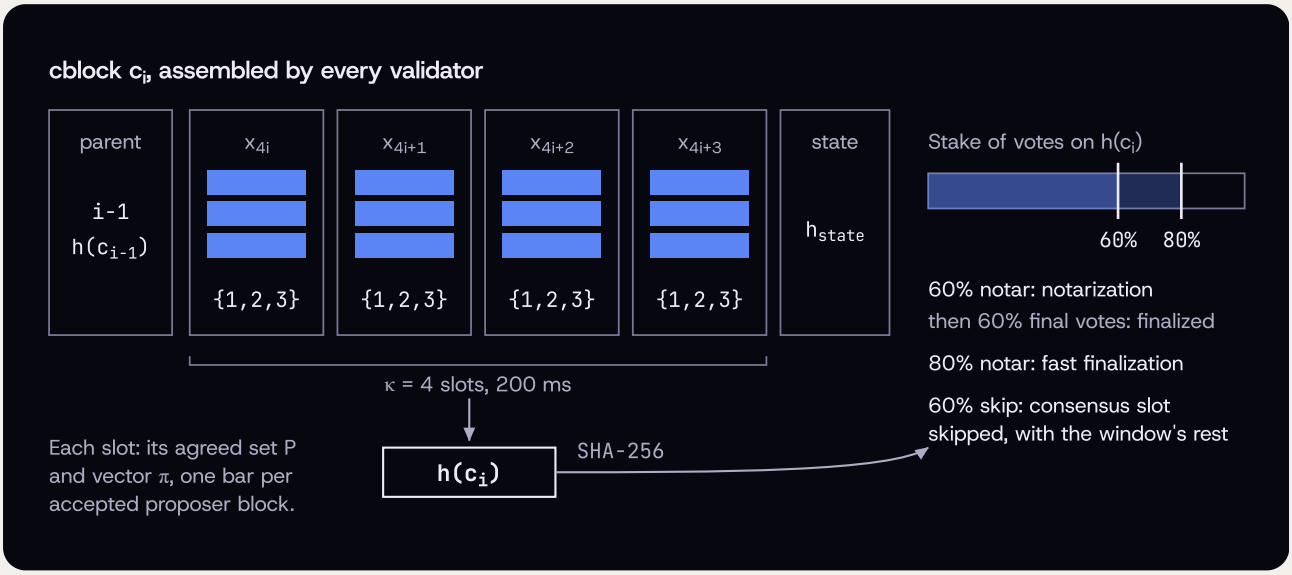



Figure 4.2 A cblock as validators assemble it: the parent reference, four xblocks each with its agreed set and attestation vector, and the state hash after the fourth. Its SHA-256 is the only thing Alpenglow sees, and the thresholds are Alpenglow's, counted by stake.

The cblock is never sent by anyone. Each validator assembles it from what it validated and executed (line 49):

$$c_i = (i_{parent}, h(c_{i_{parent}}), x_{4i}, x_{4i+1}, x_{4i+2}, x_{4i+3}, h_{state})$$

In `Network::try_vote`, $h(c_i)$ is the SHA-256 of the consensus slot $i$, the parent reference, and for each of the four slots the slot number, its agreed set $P_s$ and its attestation vector $\pi_s$, followed by the state hash after the fourth slot. Every $\pi_s$ holds the commitments of the accepted blocks, and every commitment covers its block's ciphertext and key. So $h(c_i)$ binds the cblock to its parent, to exactly one set of proposer blocks with their payloads, and to the state after executing them. The vote is `Votor::local_block` on the reference $(i, h(c_i))$ with its parent; Votor then casts the notarization vote if the parent is safe.

### 4.6.1 The attestation deadline

If $P$ does not hold all $k$ proposers, the attestation of a proposer outside $P$ may be missing, and the validator cannot tell whether that proposer is offline or its attestation is still on its way. So it executes the slots without delay, but holds back its vote for the window's first cblock until it holds the attestations of all $k$ proposers for the window's first slot, at the latest until

$$Deadline(i_0) = \text{ParentReady} + \delta_{\text{block}} + 2\delta_{\text{global}} + 2\delta_{\text{local}},$$

which on Acrux Net is 50 + 500 + 20 = 570 ms after ParentReady (lines 7, 9, 10, 46). By then the attestation of every correct proposer for that slot has arrived: its ParentReady is at most $\delta_{\text{global}}$ later than the validator's, it completes its first block $\delta_{\text{block}}$ later, attests at most $2\delta_{\text{local}}$ after that, and the attestation travels at most $\delta_{\text{global}}$. So colluding proposers cannot drop a correct co-proposer by attesting only each other; a disagreeing attestation that arrives before the deadline makes the validator vote skip. The argument is Lemma 2.2 in Chapter 7.

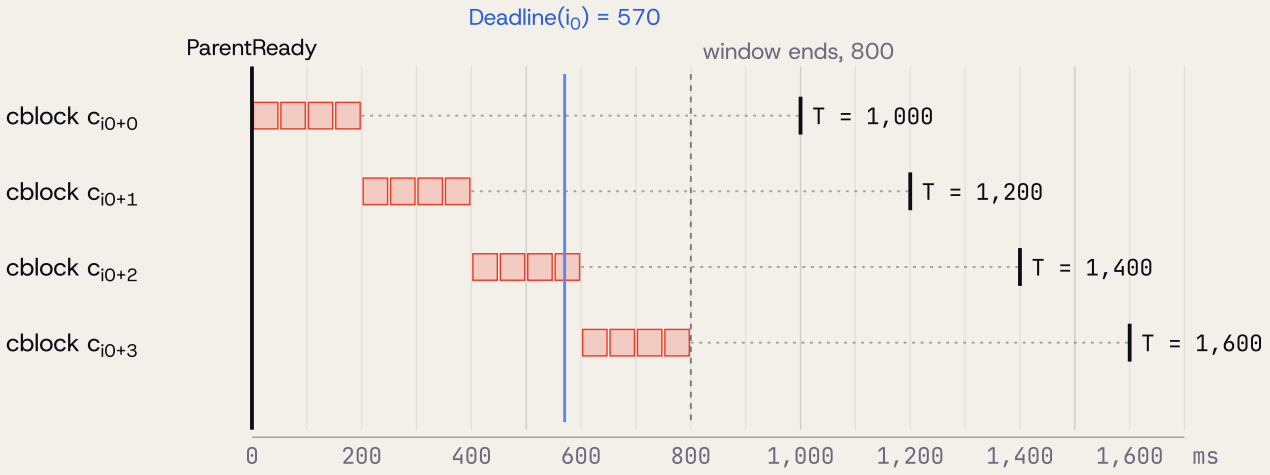



**Figure 4.3** The timeouts of a window's four cblocks after ParentReady, from $T(i)$ with the parameters of Acrux Net. Red: the four 50 ms slots each cblock covers. Each timeout falls 800 ms after its cblock's last slot ends: $\Delta_{\text{timeout}}$ plus $\delta_{\text{block}}$ for execution. Blue: the attestation deadline, which holds back only the first cblock's vote.

The wait happens once per window. The set cannot change after the first slot, so a validator that has voted for $c_{i-1}$ of the same window votes for $c_i$ as soon as it has executed it (line 47). In `Network::try_vote` the first cblock of a window waits while less than 570,000 µs have passed since the window's ParentReady and fewer than three attestations of the window's first slot are held; later cblocks wait only for the previous vote.

## 4.7 Timeouts

As in Alpenglow, a validator sets the timeouts of all consensus slots of a window when it emits ParentReady for the window's first consensus slot $i_0$. A validator needs a little more time than in Alpenglow: attestations are delayed by the hash exchange, up to $2\delta_{\text{local}}$, and execution takes up to $\delta_{\text{block}}$ once the last block and attestation of a cblock are in. The $2\delta_{\text{local}}$ come free as long as $2\delta_{\text{local}} \le \delta_{\text{global}}$, since attestations go directly to validators and save a hop against the blocks, which take two through Rotor. So only $\delta_{\text{block}}$ is added to the Alpenglow timeout [1]:

$$T(i) = \Delta_{\text{timeout}} + (i - i_0 + 1)\cdot\kappa\cdot\delta_{\text{block}} + \delta_{\text{block}}$$

after ParentReady, with $\kappa\cdot\delta_{\text{block}} = \delta_{\text{cblock}}$ = 200 ms. Acrux sets $\Delta_{\text{timeout}} = 3\delta_{\text{global}}$ = 750 ms, the least value the paper's timing assumption allows, so the four cblocks of a window time out at 1,000, 1,200, 1,400 and 1,600 ms after ParentReady. `Network::try_vote` computes the same expression in microseconds from the window's start, 750,000 + ($i$ − 4$w$ + 1)·200,000 + 50,000, where 4$w$ is the window's first consensus slot.

The three timing conditions hold with room. $\Delta_{\text{timeout}} \geq 3\delta_{\text{global}}$ is 750 ms ≥ 750 ms, and $2\delta_{\text{local}} \leq \delta_{\text{global}}$ is 20 ms ≤ 250 ms. For a window with an offline proposer to finalize its last cblock no later than a window of correct proposers, the paper asks $w\cdot\delta_{\text{block}} \geq \delta_{\text{block}} + 2\delta_{\text{global}} + 2\delta_{\text{local}}$: 800 ms ≥ 570 ms. The deadline falls while the window is still producing its twelfth slot, long before the first timeout.

## 4.8 Consensus on cblocks

New Constellation runs a single Alpenglow instance on cblocks. For Alpenglow, $c_i$ is simply the block of consensus slot $i$, identified by $h(c_i)$. Only two things differ from Alpenglow: when a cblock is valid, which is the voting rule above, and the timeouts, which leave room for execution. Votes and certificates are Alpenglow's, on $h(c_i)$ [1, 3]. A NewCon window of $w$ = 16 slots is an Alpenglow leader window of $w/\kappa$ = 4 consensus slots, and one Alpenglow instance decides one chain of cblocks, at most one per consensus slot, each decision covering up to $k$ proposer blocks per slot.

`acrux-consensus` is Votor, the voting half of Alpenglow, as a per-validator state machine over cblock references. It counts stake by distinct validator and checks every certificate's votes and stake before accepting it. Its inputs are a locally executed cblock ( `Votor::local_block` ), a timeout ( `Votor::timeout` ), and votes and certificates from the network ( `Votor::receive` ); its outputs are votes and certificates to broadcast, finalizations, and ParentReady events.

Table 4.2 Votes and certificates of Votor as acrux-consensus implements them. Thresholds are shares of total stake.

| Vote or certificate | Cast or formed when | Effect |
|---|---|---|
| Notarization vote | the cblock is executed locally and its parent is safe; at most one initial vote per consensus slot | the validator's notar-or-skip choice for the slot |
| Skip vote | $T(i)$ passed without a notarization vote, or the window is already skipping | the other initial choice; exclusive with notarization |
| Notar-fallback vote | 40% initial notarization stake on that cblock, or 20% on it and 60% notarization and skip together | lets a split first round still certify a cblock |
| Skip-fallback vote | 40% of initial stake once the largest notarization target is set aside | lets a split first round still skip |

| Vote or certificate | Cast or formed when | Effect |
|---|---|---|
| Notarization certificate | 60% notarization votes for $h(c_i)$ | $c_i$ notarized; a final vote follows |
| Notar–fallback certificate | 60% notarization and notar–fallback votes | $c_i$ certified, a parent for the next window |
| Fast finalization | 80% notarization votes | $c_i$ finalized in one round |
| Final vote | a notarization certificate, in a slot where this validator voted to notarize and cast no skip or fallback | the second round of the slow path |
| Finalization certificate | 60% final votes | $c_i$ and its ancestors finalized |
| Skip certificate | 60% skip and skip–fallback votes | consensus slot *i* skipped |

4.8.1

### Safe to notarize, safe to skip

Votor casts its notarization vote only on a cblock whose parent is safe. For the first cblock of a window the parent must be certified, by a notarization, notar–fallback or finalization certificate, and every consensus slot between the parent and the cblock must hold a skip certificate. For a later cblock of the window the parent must be certified or be the cblock this validator itself voted to notarize, the preceding one. Finalized history is never contradicted: a cblock that does not extend the finalized one is not voted for. The fallback votes follow Alpenglow's safe–to–notar and safe–to–skip conditions as stake thresholds over the initial votes of the slot: a validator that saw enough stake notarize one cblock may add a notar–fallback for it, and one that saw enough stake fail to agree on any single cblock may add a skip–fallback, so a slot whose first round split still reaches a certificate. Fallback votes are cast at most once per target, and a final vote, once cast, closes fallback voting for that slot.

4.8.2

### ParentReady

ParentReady for the first consensus slot of the next window is emitted, as in Alpenglow, once the validator holds a certificate (other than skip) for some earlier cblock and a skip certificate for every consensus slot after it up to the window boundary. `Votor::ready_parent` takes the latest such cblock, never one below the finalized cblock, and emits ParentReady with it as the parent. That one event does two things: it starts the next window's proposers (Section 3.2.1) and it sets every validator's timeouts and deadline for that window. A proposer that receives a certificate forwards it, with the signed votes it was formed from, to the proposers of the certificate's window and of the next, so co–proposers see the same certificates within $\delta_{\text{local}}$ of each other.

### 4.8.3 Execution before voting

> **A notarization certificate on $h(c_i)$ certifies the proposer blocks and the state they produce, because all proposer blocks of a consensus slot are decided at once.**

A validator that votes notarize knows the exact set $P$ and has executed exactly those blocks, so the proposer blocks of a consensus slot are decided and executed as a single Alpenglow block is. The state hash cannot come from a proposer, since the state after a slot depends on every proposer's block and no proposer knows the others' blocks before validators do; it is part of the cblock the validators assemble. New Constellation thus executes before voting, though not before block production ends. Designs that decide each proposer's block independently must execute after consensus, and their votes cannot certify the resulting state [1]. On Acrux Net a cblock finalizes at p50 324 ms and p99 398 ms after its first slot (Chapter 9).

## 4.9 Skips

As in Alpenglow, a skipped cblock skips the rest of its window, and the next window starts right away. A validator votes skip for consensus slot $i$ when its timeout passes before the cblock is voted, or as soon as one of its slots shows disagreeing attestation vectors, an equivocation or an invalid block. In `Network::try_vote` the first skip vote of a window records the consensus slot it skipped from, and every later cblock of that window gets a skip vote too; the skipped slots are reported with their reason, disagreeing vectors or a missing attestation, and with the vectors themselves. Once a skip certificate for a consensus slot of a window arrives, `try_execute` executes no slot from it on. When the window's remaining consensus slots are certified skipped, ParentReady for the next window follows from the last certified cblock, and the next hub's proposers start.

A skip never loses a transaction for good. The skipped blocks are never executed, so nothing in them took effect, and each proposer of the skipped window hands the transactions and fragments it still holds to the proposer in the same role of the next window (Section 3.5).

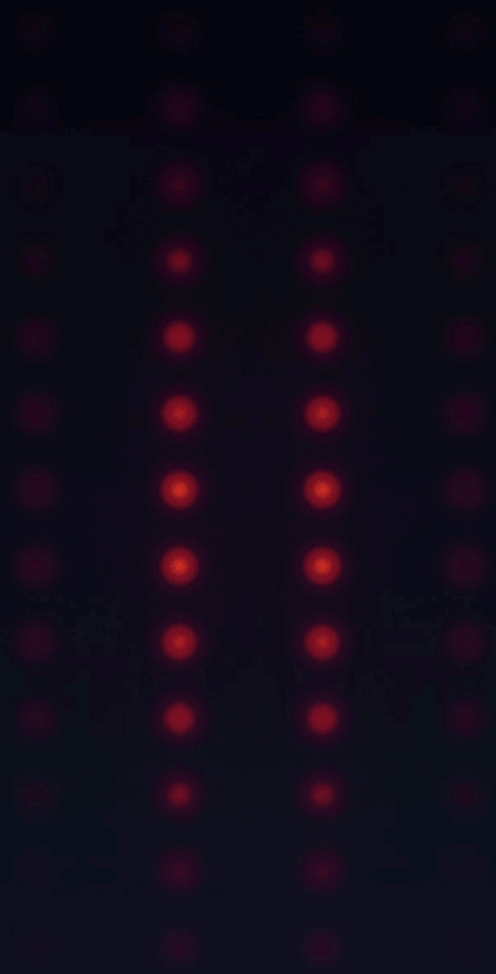
## 4.10 A window with a crashed proposer

A proposer that is crashed from the start of its window does not cause a skip. All its co-proposers attest it late, so the attestations agree on the other two, $P = \{1, 2\}$ is a strict majority, and every slot of the window is agreed and executed on two blocks. Execution runs without delay. Only the first vote of the window waits, for the attestation that never arrives, until $Deadline(i_0)$ = 570 ms after ParentReady; the later three cblocks are voted as soon as they

are executed, because the set cannot change within the window. Since 570 ms is less than the 800 ms of the window, the last cblock of such a window finalizes no later than it would with three correct proposers [1].

In a 60 s run of Acrux Net with one proposer silent, slot 32 was agreed on $P = \{1, 2\}$ and the window's first notarization vote left 570.2 ms after the window began, at the deadline; 261 cblocks were finalized and none skipped, and all 1,052 state-hash heights compared matched across the twelve validators. The 120 s run of Section 3.6 finalized 526 cblocks and skipped none. The Fault Lab's crash scenario replays the same fault on demand and records the agreed set as evidence (Chapter 10); the safety and liveness of the whole construction under Alpenglow's model are proved in Chapter 7.

M 4 TRACKS · 2,000,000 CU EACH
BASE FEE 5,000 · INCLUSION 1,000 LAMPORTS
RESERVE $\rho$ 0.001 ACRUX



5

# Ordering, threshold transactions and fees

The merge has no author: one deterministic order by priority fee per compute unit, four execution tracks, a transaction split so no single proposer can read it, and fees that pay every proposer that carried it.

Once a slot is agreed on a set $P$, its proposers have no further say. Every validator merges the transactions of the accepted blocks into one order, fills a fixed virtual schedule with them and charges fees by rules that read only the transactions themselves.

## 5.1 The merge

A validator reaches the merge with the decrypted payloads of every proposer in the agreed set $P$ of a slot (Chapter 4). The proposers outside $P$ are ignored. What remains to decide is which of these transactions execute and in what order, and the paper decides both without any input from the proposers: the merge depends only on the transactions, never on which proposer carried a transaction or when within the slot it arrived [1]. The slot a transaction lands in is what counts.

The merged and selected transactions of slot $s$ form its execution block $x_s$. The xblocks are executed one after the other, each on the state left by the previous one, and the first slot of a window on the state after its parent cblock. This is the call $\texttt{Execute}(\sigma, \texttt{Merge}(s, (T_q)_{q \in A}))$ of Algorithm 2, line 38, which the paper fills with Algorithm 4.

### 5.1.1 Removing duplicates

A user may send one transaction to several proposers ($n > 1$), or send it again to the proposers of a later window, so the merge first removes duplicates [1]:

- **Plain transactions** are identified by the hash of their signed message. Copies with the same message are the same transaction, and one copy is kept.
- **Private transactions** ($t > 1$) are identified by their transaction ID, the commitment in their public header. The fragments that different proposers included are collected under this ID, and the transaction is kept once.
- **Earlier slot wins.** If the same transaction was included in different slots, only the copy in the earlier slot counts. A transaction already executed in an earlier slot is recognized by its message hash and dropped.
- **Incomplete private transactions** wait in a fragment cache until their $t$-th fragment is accepted, and then execute in that slot (Section 5.3.2).

Acrux implements the merge in `SvmExecutor::execute_slot` (`backend/node/crates/acrux-execution/src/execution.rs`). A plain transaction's ID is the SHA-256 hash of its serialized message, the same value a private transaction's header commits to, so both kinds meet in one map keyed by ID. Every copy is checked again at this boundary: its signed bytes are revalidated, and its public header must equal the header derived from the signed message, so a header can redirect neither fees nor writable-account locks. Copies with the same ID merge into one entry whose *source proposers* are the union, sorted and deduplicated, of the proposers that

carried it; that set decides who is paid (Section 5.4.1). The executor keeps every executed ID for as long as the blockhash it referenced stays valid, 1,200 slots, so any later copy of an executed transaction is dropped before it reaches the order.

### 5.1.2 The order

After duplicates are removed, the transactions of the slot are ordered by priority, as in Legacy Constellation [2]. `SortTxs(txs)` sorts by

1. priority fee per compute unit *prio*, decreasing,
2. compute limit *cu*, increasing,
3. transaction hash, increasing.

The order is total, since two different transactions have different hashes, and it is a function of the transactions alone. No proposer can move a transaction it carries ahead of one it does not, and getting ahead of a transaction means outbidding it. In Acrux, `sort_transactions` in `selection.rs` is this comparator over `(priority_fee_per_cu, compute_limit, id)`. The priority fee is declared in micro-lamports per compute unit, as on Solana, and the fee charged is $prio \cdot cu \,/\, 10^6$ lamports, rounded up (`priority_fee`).

The paper names round robin and application-defined orders as alternatives; Acrux runs priority order.

The paper notes that other orders fit the same slot: round robin among the proposers, or an order each application defines for its own transactions, such as a batch auction per market. Application-defined ordering must stay composable when a transaction touches several applications; Canidio proposes following an order whenever all affected applications agree on it and priority order otherwise [22], and the paper adds a variant in which each application defines a priority function and a transaction is valid only if every application it touches assigns it the same priority [1].

## 5.2 Slot capacity and Algorithm 4

Every transaction declares a compute limit *cu* and the set *acc* of writable accounts it touches. If its execution exceeds *cu*, it is canceled and still pays its fees. A slot is limited by a *virtual parallel execution schedule* of $m$ execution tracks, each of which may use at most $b_{cu}$ compute units; $b_{cu}$ thus bounds the time it takes to replay the slot [1]. The paper suggests $m = 4$. Acrux runs $m = 4$ tracks of $b_{cu} = 2{,}000{,}000$ compute units, 8,000,000 per 50 ms slot (`TRACKS` and `TRACK_CAPACITY_CU` in `selection.rs`, configurable through `ACRUX_BCU`).

Validators fill the schedule deterministically, walking down the merged order:

- A transaction is placed at the earliest time on any track at which it does not overlap an earlier transaction, in the merged order, that writes one of the same accounts, and at which it still finishes by $b_{cu}$. If several tracks are free at that time, the track with the smallest index is used.

— This can leave gaps. A later transaction may fill a gap, as long as no conflicting earlier transaction starts after it.

Algorithm 4 states the procedure. The set `spaces` holds the free intervals as triples (start, end, track); `scheduled` holds every placed transaction as (position, length, accounts, track). For each transaction in order, the algorithm takes free intervals one at a time (line 13) and tries the candidate positions in that interval: the interval's start and the end of every scheduled transaction (lines 15 to 17). At a candidate position, $C$ collects the scheduled transactions that overlap the transaction's run (line 18), and `FindSpace` rejects the position if the run does not fit before the interval's end or if it writes an account that an overlapping transaction writes (line 39). On success the interval is split into what remains before and after the transaction (lines 43 to 47), and the transaction is scheduled.

**Algorithm 4** Transaction selection

```
1  function SelectTxs(txs)
2    txList ← [ ]
3    spaces ← {(0, b_cu, 1), …, (0, b_cu, m)}   ▷ free locations: start, end, track
4    scheduled ← ∅   ▷ scheduled txs: position, length, accounts, track
5    txs ← SortTxs(txs)   ▷ by priority
6    for tx ∈ txs do   ▷ iterate in order
7      newSpaces ← ∅
8      placed ← false
9      if VM.chargePriorityFee(tx) failed then
10       placed ← true   ▷ skip tx (still charge for inclusion)
11     end if
12     while not placed and spaces ≠ ∅ do
13       (start, end, ℓ) ← min spaces
14       spaces ← spaces \ {(start, end, ℓ)}
15       stxs ← sortBy(scheduled ∪ {(0, 0, ∅)}, (pos, len, acc) ↦ pos + len)
16       for (pos, len, _, _) ∈ stxs do
17         newPos ← max{start, pos + len}
18         C ← {(p, l, _, _) ∈ stxs : p + l > newPos ∧ p < newPos + tx.cu}
19         busyAcc ← {a : (_, _, a, _) ∈ C}
20         S ← FindSpace(tx, newPos, (start, end, ℓ), busyAcc)
21         if S ≠ {(start, end, ℓ)} then
22           scheduled ← scheduled ∪ {(newPos, tx.cu, tx.acc, ℓ)}
23           txList ← txList ∪ {tx}
24           placed ← true
25           break
26         end if
27       end for
28       newSpaces ← newSpaces ∪ S
29     end while
30     if placed then
31       VM.chargeInclusionFee(tx)
32     end if
33     spaces ← spaces ∪ newSpaces
34   end for
35   txList ← SortTxs(txList)
36   return txList
```

```
37  end function
38
39  function FindSpace(tx, pos, (start, end, ℓ), busyAcc)   ▷ returns new spaces
40    if end - pos < tx.cu or tx.acc ∩ busyAcc ≠ ∅ then
41      return {(start, end, ℓ)}
42    end if
43    S ← ∅
44    if pos > start then
45      S ← S ∪ {(start, pos, ℓ)}
46    end if
47    if pos + tx.cu < end then
48      S ← S ∪ {(pos + tx.cu, end, ℓ)}
49    end if
50    return S
51  end function
```

In acrux-node: lines 1 to 8 and 12 to 50 in `sort_transactions` and `select_with_fees`, `backend/node/crates/acrux-execution/src/selection.rs`; the fee calls of lines 9 to 11 and 30 to 32 in `SvmExecutor::execute_slot` and `SvmExecutor::charge_inclusion`, `execution.rs`.

Line 9 charges the priority fee before any placement is tried. If the fee payer cannot pay it without touching its reserve, the transaction is marked as handled without being scheduled, and its inclusion fee is still charged (Section 5.4.3). Line 35 sorts the selected list again, so the xblock executes in merged order, whatever track each transaction was placed on.

### 5.2.1 Selection in acrux-node

`select_with_fees` takes the merged candidates, the track capacity and a fee callback that is line 9. It keeps the paper's data: the free intervals as `Space` values, the scheduled transactions with their writable accounts, and the candidate positions, which are 0 and the end of every scheduled transaction (line 15). One point is made explicit. Line 13 takes the minimum free interval, and the prose of the paper says a transaction goes to the earliest time on any track, the lowest track on a tie. The node evaluates every free interval's first valid position and then chooses the earliest position, breaking ties by the lowest track, which is the rule the prose and the paper's Figure 5 state. It returns the placements, the remaining gaps, and the selected and rejected IDs; the gateway serves all of them on every slot's page, so the schedule of any slot of Acrux Net can be read after the fact.

### 5.2.2 A worked example

The paper illustrates Algorithm 4 with thirteen transactions $t_1$, …, $t_{13}$, in decreasing priority, over accounts A to G. Its figure fixes the accounts and the order but no compute scale. Acrux's test `figure_five_contention_and_gap_filling` gives the transactions compute limits of 4, 3, 6, 5, 3, 3, 4, 3, 5, 4, 2, 4 and 4 units and a track capacity of 14 units. Figure 5.1 is what `select_with_fees` returns for this input.

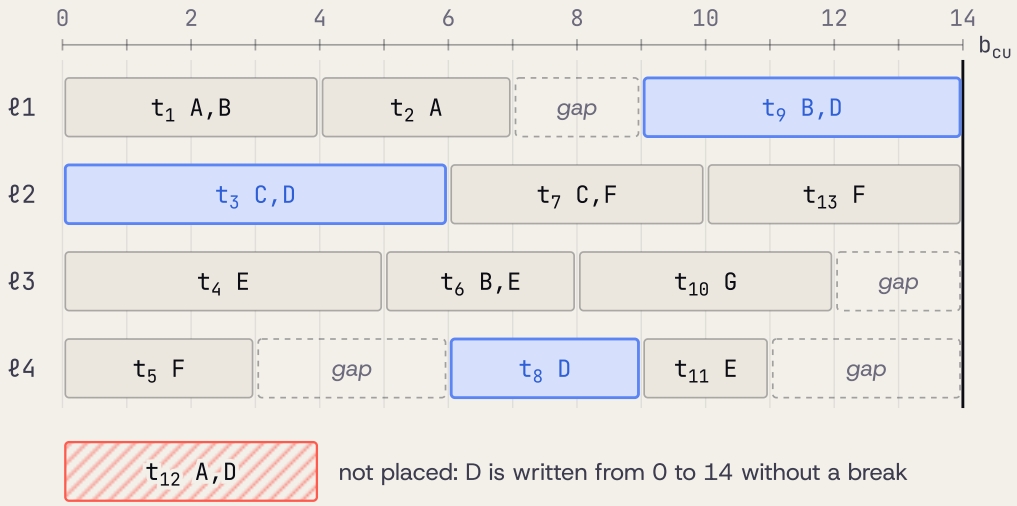



**Figure 5.1** The paper's thirteen-transaction instance through `select_with_fees`, with a capacity of 14 units per track. Each block is $t_i$ with the accounts it writes; the writers of account D are drawn in blue. $t_8$ waits for $t_3$, which leaves a gap on track 4, and $t_{12}$ cannot be placed, since D is written until the end of the slot.

Walking the order: $t_1$, $t_3$, $t_4$ and $t_5$ touch disjoint accounts and start at 0 on tracks 1 to 4, while $t_2$, which writes A like $t_1$, takes the earliest conflict-free time, 4, on track 1. $t_6$ writes B and E, so it waits for both $t_1$ and $t_4$ and starts at 5. $t_8$ writes D and must wait for $t_3$ until 6; the earliest free track at 6 is track 4, which leaves the gap from 3 to 6 behind $t_5$. $t_9$ writes B and D and waits for $t_8$ until 9, taking the end of track 1. The later and shorter $t_{10}$, $t_{11}$ and $t_{13}$ fill what is left on tracks 2 to 4. $t_{12}$ writes A and D, and D is written by $t_3$, $t_8$ and $t_9$ from 0 to 14 without a break, so $t_{12}$ is not selected, while $t_{13}$, which comes after it, is. Twelve transactions are selected and one is rejected, which is what the test asserts.

### 5.2.3 Priority inversion

Filling gaps can invert the priority order of two conflicting transactions. If $tx_1$ has a higher priority than $tx_2$ but conflicts with many other transactions, including transactions on its own track, $tx_1$ may fit only late in the schedule, while $tx_2$, with few conflicts, fits an earlier gap and executes strictly before $tx_1$. The paper accepts this deliberately: $tx_1$ has used its priority fee for its position relative to the transactions it conflicts with, most transactions care mainly about their most congested accounts, and forbidding such inversions would let a single transaction destroy all the parallelism of the slot [1].

### 5.2.4 Deterministic, not optimal

The greedy schedule does not always fit the most transactions into a slot. This is unavoidable: already for transactions that touch disjoint accounts and whose compute limits add up to exactly $m \cdot b_{cu}$, deciding whether they all fit on the $m$ tracks is NP-complete [1]. What the schedule has instead is the property consensus needs. It is deterministic and follows the merged order, so all validators select the same transactions, and higher-priority transactions

are scheduled first. This is assumption (D) of the correctness proof (Chapter 7), and in Acrux it is checked on every slot: validators compare state hashes, 10,564 comparisons in the 600 s run of Acrux Net with no mismatch (Chapter 9).

### 5.2.5 Execution

The virtual schedule decides which transactions execute in a slot, not how. Actual execution may be faster and adaptive: if a transaction uses fewer compute units than it declared, later transactions may start earlier. The one requirement is that every pair of transactions writing a common account executes in merged order; then the result does not depend on how execution is parallelized [1]. Acrux executes the selected list in merged order through the Solana Virtual Machine, one transaction after the other, and commits the resulting accounts into the state hash of the slot (Chapter 8).

## 5.3 Threshold transactions

The merge protects the order of a slot once it is fixed. Threshold transactions protect a single transaction while the slot is being built. A user sends each transaction to a set of $n$ proposers of the target slot, $1 \le n \le k$, with a threshold $1 \le t \le n$: a *$t$-of-$n$ transaction* [1]. The two numbers buy different things.

- **A larger $n$ buys censorship resistance.** The transaction survives $n - t$ receivers that censor it or whose blocks are not accepted.
- **A higher $t$ buys privacy.** The content stays hidden from any $t - 1$ colluding receivers, until $t$ fragments have been revealed.

On Acrux Net, $k = 3$, and the SDK and the composer on `/send` offer the choices of Table 5.1, with 2-of-3 as the default: `await acrux.send(tx, { n: 3, t: 2, signer })`.

Table 5.1 The $t$-of-$n$ choices with $k = 3$. Inclusion fees are 1,000 lamports per included copy or fragment.

| n | t | What it buys | Survives | Hidden from | Inclusion fees |
|---|---|---|---|---|---|
| 1 | 1 | One proposer carries it in the clear. Cheapest. | 0 drops | nobody | 1,000 |
| 2 | 1 | Two copies; either proposer lands it. | 1 drop | nobody | 2,000 |
| 3 | 1 | Three plain copies. | 2 drops | nobody | 3,000 |
| 2 | 2 | Split in two; no single proposer can read it. | 0 drops | any 1 | 2,000 |
| **3** | **2** | **Split three ways; any two rebuild it, no one reads it alone.** | 1 drop | any 1 | 3,000 |
| 3 | 3 | All three fragments needed. Most private. | 0 drops | any 2 | 3,000 |

5.3.1

### Splitting a transaction

For $t = 1$, the plain signed transaction goes to every receiver, and the merge keeps one copy (Section 5.1.1). For $t \geq 2$, the client follows Krawczyk's secret sharing made short [11]:

1. Encrypt the transaction with a fresh symmetric key.
2. Erasure code the ciphertext into $n$ fragments, any $t$ of which reconstruct it.
3. Split the key into $n$ Shamir shares with threshold $t$.
4. Send each receiver one fragment, one key share and a public header signed by the fee payer.

Krawczyk's construction is what keeps the fragments short: only the 32-byte key is secret-shared, so each fragment carries about $1/t$ of the ciphertext rather than a full-size share of it. The public header carries everything that validity, merging and selection use, so the fragments can be validated, deduplicated, rate-checked and charged before anyone can read the transaction.

Table 5.2 The public header of a threshold transaction, signed by the fee payer.

| Field | Used for |
|---|---|
| `id` | The transaction ID: the SHA-256 hash of the transaction's message, which identifies it across fragments and against plain copies |
| `feePayer` | The account charged the inclusion, base and priority fees |
| `priorityFeePerCu` | Its place in the merged order, in micro-lamports per compute unit |
| `computeLimit` | Its length in the slot's execution schedule |
| `writableAccounts` | Its conflicts in the schedule |
| `targetSlot` | The slot it was sent for; no fragment is accepted before it |

Acrux fixes the constructions the paper leaves open ( `backend/node/crates/acrux-execution/src/threshold.rs` ). The cipher is ChaCha20-Poly1305 under a fresh 32-byte key, with the serialized header as associated data, so a fragment cannot be moved under a different header. The fragments are Reed-Solomon shards over GF(256), $t$ data shards and $n - t$ parity shards, each of the ciphertext length divided by $t$, rounded up. The key shares are Shamir shares over GF(256), byte by byte, at the points $x = 1, \ldots, n$. The byte layouts of fragment and share are in Appendix D. Figure 5.2 follows the default 2-of-3 split.

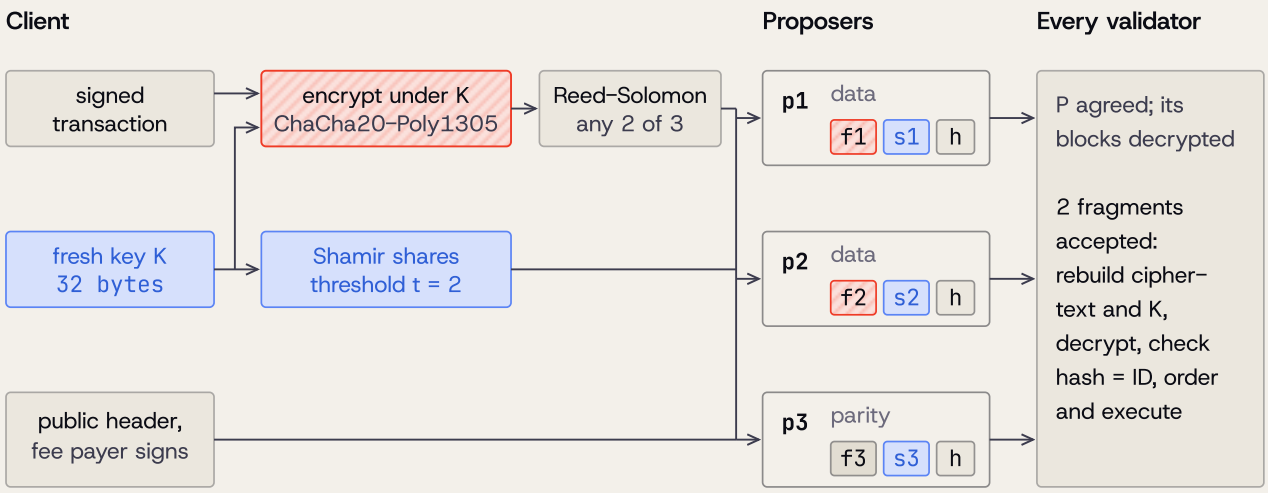



**Figure 5.2** A 2-of-3 transaction. The client seals the transaction under a fresh key K, codes the ciphertext into three fragments of which two are data and one parity, and splits K into three Shamir shares with threshold 2. Each proposer of the target slot receives one fragment, one share and the signed header; one of each reveals nothing. Validators rebuild and execute the transaction once two fragments are accepted.

A fragment submission is checked before a proposer may include it, and again when a block carrying it is decoded (`FragmentSubmission::verify`, `VerifiedFragment::revalidate`): the header signature must verify under the fee payer's key; $2 \le t \le n \le 3$; the compute limit is positive and at most 1,400,000; the ID is a 32-byte hash and every account a valid key; and the fragment length matches the declared ciphertext length and $t$. Each fragment names its receiver, and a validator accepts it only from the block of that receiver, one fragment per receiver and transaction.

### 5.3.2 The fragment cache and decoding

A private transaction can be decoded only once $t$ of its fragments have been accepted, and these may land in different slots. Validators therefore keep received fragments in a cache indexed by transaction ID. As soon as the $t$-th fragment is received in some slot, the transaction is decoded and executed in that slot, ordered like any other transaction of the slot. If fewer than $t$ fragments have been accepted after an expiry period, the cached fragments are discarded [1].

In Acrux this is `ThresholdCache`. A fragment enters the cache only after the block that carries it belongs to the agreed set $P$ of its slot (`accept_at`). All fragments under one ID must carry the same header, $n$, $t$, nonce and ciphertext length; a second, different fragment for the same index is rejected. When the $t$-th distinct fragment arrives, the node rebuilds the ciphertext with Reed-Solomon, interpolates the key from the $t$ shares, decrypts, deserializes the Solana transaction, verifies its signatures and checks that the hash of its message equals the header's ID (`validate_message_commitment`). Only then does the transaction join its slot's merge, under the header's priority, with the proposers whose fragments rebuilt it as its carriers. A decoded ID is never decoded again, and a later fragment for it changes nothing.

The **fragment expiry** is 8 slots, 400 ms, counted from the slot of the first accepted fragment ( `ACRUX_FRAGMENT_EXPIRY` ). A fragment arriving in a later slot is refused, and `ThresholdCache::expire` discards every incomplete entry past its expiry at the start of each slot. Because the cache is part of what a validator executes, it is checkpointed with the state: a window that builds on a different parent restarts from the cache as it stood after that parent.

**Table 5.3** The states Acrux reports for a threshold transaction, on its page at `/tx/[id]` and through the SDK.

| State | Meaning |
|---|---|
| `sent` | Fragments and shares delivered to the receivers |
| `partly included` | Some fragments accepted, fewer than $t$ |
| `executed` | The $t$–th fragment accepted; decoded, ordered and executed in that slot |
| `finalized` | Its cblock finalized |
| `failed` | Executed and failed, or decoded to an invalid transaction; fees charged |
| `expired` | Fewer than $t$ fragments accepted within the fragment expiry |

### 5.3.3 How long a transaction stays hidden

Hiding by threshold lasts until $t$ fragments have been revealed, which is when the blocks that carry them are decrypted. If all receivers include their fragments in the target slot, the transaction is revealed together with the slot that executes it: parties that have now seen $t$ fragments can only act in later slots, which are ordered after it, so they can back–run the transaction but not front–run it [1]. A receiver that holds its fragment back until the others are revealed could decode the transaction before carrying its own fragment; the paper bounds this by the expiry of the cache, and Acrux keeps that expiry at 8 slots. A slot that is skipped after its fragments were disseminated reveals the transaction without executing it, and a resent copy goes into a later slot as a new send.

The two hidings compose. Block hiding (Lemma 2.3) keeps every proposer from seeing the others' blocks before its own is fixed; threshold hiding keeps the receivers of a transaction from reading it out of their own inboxes. The Fault Lab's *Private front–run* scenario runs a 2–of–3 transfer against a receiver that tries to read it from the one fragment it holds: one of the two fragments it would need, so it rebuilds neither the ciphertext nor the key, and the transaction executes in its slot after every block of the slot is fixed (Chapter 10).

## 5.4 Fees

The paper keeps the fees of Solana and adds one: a base fee, part of which is burned; a priority fee, which also decides the order; and an *inclusion fee* that every included copy or fragment of a transaction pays to the proposer

whose accepted block carries it [1]. It leaves the details of the fee mechanism open, and Acrux sets them as Table 5.4 shows.

**Table 5.4** The three fees on Acrux Net (`Fees::default`, `execution.rs`).

| Fee | Charged | Goes to | Amount |
|---|---|---|---|
| Base fee | Once, when the transaction is selected and executes | 50% burned; 50% split among the proposers that carried it | 5,000 lamports per signature |
| Priority fee | Once, at selection (Algorithm 4, line 9) | Split among the proposers that carried it | *prio·cu* / $10^6$, set by the sender |
| Inclusion fee | For every included copy or fragment | The proposer whose accepted block carries that copy or fragment | 1,000 lamports |

5.4.1

### Paying the proposers that carried it

With a single leader there is one block producer to pay. A slot of New Constellation can carry the same transaction in several accepted blocks, as copies of a plain transaction sent with $n > 1$ or as the fragments of a threshold transaction. Acrux pays them by two rules.

- **Inclusion per copy.** Each included copy or fragment pays 1,000 lamports to the proposer that carried it, whether or not the transaction is selected, executes, or was already executed in an earlier slot. This gives a proposer a reason to include a transaction even when it does not execute in the end, and it makes redundancy cost the user: an $n$–way send pays $n$ inclusion fees [1].
- **Equal shares of the rest.** The unburned half of the base fee and the whole priority fee go in equal shares to the proposers that carried the transaction: every proposer whose accepted block holds a plain copy, or, for a threshold transaction, the proposers whose $t$ fragments rebuilt it; a fragment accepted after the decode earns its inclusion fee only. The remainder lamport goes to the lowest proposer index among them (`SvmExecutor::distribute`).

The two rules mirror the merge. The order of a slot ignores which proposer carried a transaction, so the fees that come with the transaction's execution do not single out one carrier either: every proposer that delivered it in an accepted block shares them, and none gains by being first or by keeping a copy out of the others' blocks. A fragment of a threshold transaction earns its inclusion fee like a copy, and the $t$ fragments that rebuild the transaction share its execution fees, since each of them is needed.

Table 5.5 shows what this comes to for the sends measured on Acrux Net: ten sends of each kind, a 0.001 ACRUX transfer with one signature and no priority fee, through `@acrux/sdk`.

Table 5.5 Fees in lamports of one-signature sends with no priority fee. Fees and medians from submit as measured on Acrux Net ( `backend/sdk/test/cost.mjs` ); the split follows the rules above.

| Send | Base | Inclusion | Total | To proposers | Executed | Finalized |
|---|---|---|---|---|---|---|
| Plain, to 1 proposer | 5,000 | 1,000 | 6,000 | 3,500 to the carrier; 2,500 burned | 339 ms | 580 ms |
| Plain, to all 3 proposers | 5,000 | 3,000 | 8,000 | 1,834, 1,833, 1,833; 2,500 burned | 342 ms | 585 ms |
| 2-of-3 threshold | 5,000 | 3,000 | 8,000 | 2,250 to each of the two proposers whose fragments rebuilt it, 1,000 to the third; 2,500 burned | 339 ms | 585 ms |

A 2-of-3 send costs the same as three plain copies, and its median time from submit to execution, 339 ms, is that of a plain send to one proposer.

### 5.4.2 The fee-payer reserve

Proposers build their blocks on an outdated view of the chain. A proposer may include a transaction whose fee payer can no longer pay, or several transactions of the same fee payer that together exceed its balance, and it cannot see what its co-proposers include in the same slot. To make sure that every inclusion fee can be paid, the paper adopts the reserve rule of Legacy Constellation [1, 2]:

- Every fee payer account keeps a small reserve $\rho$, which can only be withdrawn by closing the account, and closing takes effect only two windows later.
- A proposer includes transactions only once it has executed all slots before the previous window.
- A proposer includes transactions of a given fee payer with inclusion fees of at most $\rho/(2k)$ per window.

The inclusions that a proposer cannot yet observe therefore stem from at most two windows, the previous one and the current one, so from at most $2k$ proposers, and together they never exceed the reserve. Validators check the rate, and a proposer block that exceeds it is invalid. Acrux sets $\rho$ = 0.001 ACRUX, 1,000,000 lamports, as the paper suggests; with $k$ = 3 the rate is 166,666 lamports per fee payer, proposer and window, which is 166 inclusions at 1,000 lamports ( `PER_PROPOSER_WINDOW_LIMIT` ).

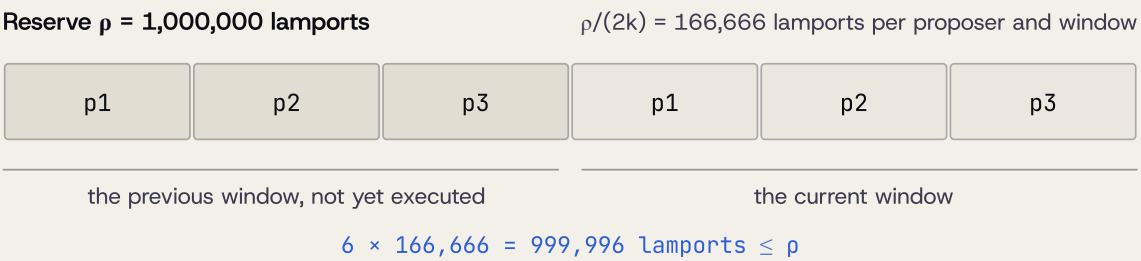



Figure 5.3 Why $\rho/(2k)$ is enough. The inclusions a proposer cannot yet see come from the three proposers of the previous window and the three of the current one; each charges a fee payer at most 166,666 lamports per window, so all six together stay within the reserve.

In the node, the rate is enforced where blocks become payloads. When a validator executes an agreed slot, `SvmExecutor::validate_inclusions` sums the inclusion fees of the slot's blocks per fee payer and proposer, adds what the same proposer already charged that fee payer in the window, and declares the slot invalid if any sum exceeds $\rho/(2k)$ or a fee payer cannot cover its debits. `charge_inclusion` then charges each copy and fragment and records the window's running total; totals older than two windows are dropped. The 1,000,000-lamport reserve and the 166-inclusion limit are covered by the executor's test `reserve_and_per_window_inclusion_rate`.

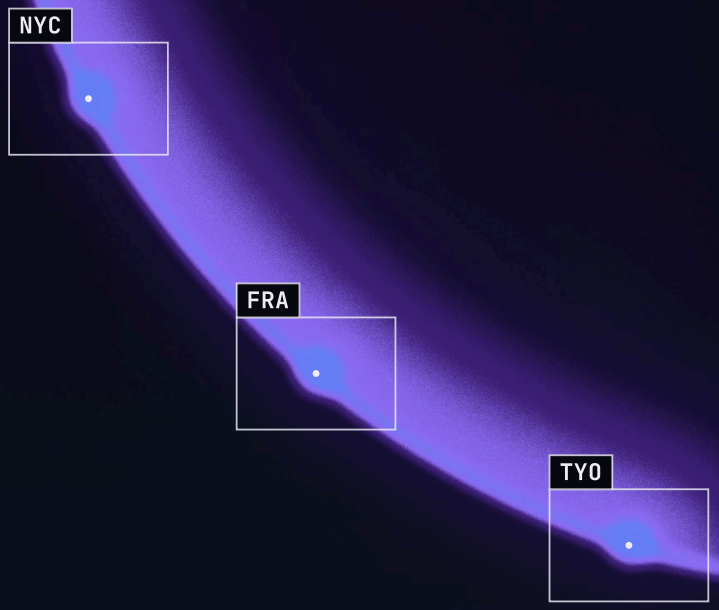
### 5.4.3 Charging in selection

During the selection (Algorithm 4), the three fees meet the reserve in a fixed order [1]:

1. The inclusion fee is always charged, for every included copy or fragment, even if the balance drops below the reserve. In Acrux it is charged when the agreed blocks are taken in, before the merge.
2. A transaction is selected only if its priority fee can be charged without touching the reserve. Otherwise it is skipped, and its inclusion fees stand.
3. A transaction whose execution would touch the reserve fails, and pays its fees. Acrux checks the fee payer's balance after simulation and records the failure as `Execution would spend the fee-payer reserve`. The base fee, too, is charged only above the reserve; a transaction that cannot pay it fails.

To send on Acrux Net, a fee payer therefore holds at least the 0.001 ACRUX reserve plus the fees of what it sends. The faucet on `/send` funds a browser key with test ACRUX for exactly this.

$\alpha$ 1/2 · $n_{min}$ 4 · $\rho_{min}$ 1%
WINDOWS OF 16 SLOTS · 800 MS
PROPOSER STAKE CAPPED AT 1/K OF A HUB




6

# Hubs and the proposer schedule

Proposers that author a slot together must sit a few milliseconds apart. Hubs make that a fact on chain, and Algorithm 3 turns hubs and stake into windows.

The proposers of a window exchange their commitments within $2\delta_{\text{local}}$, so they must sit a few milliseconds apart. New Constellation draws them from hubs of co-located validators, and Algorithm 3 turns hubs and stake into the windows of an epoch.

## 6.1 Why proposers are co-located

The hash exchange of Chapter 3 gives each proposer $2\delta_{\text{local}}$ to receive the commitments of its co-proposers: up to $\delta_{\text{local}}$ because a co-proposer may have started its slot later, and another $\delta_{\text{local}}$ for the message itself. A co-proposer whose commitment misses that window is attested late, and its block is lost. With $\delta_{\text{local}}$ = 10 ms, the proposers of a window must reach each other within a few milliseconds, which in practice means the same city. This requirement has far-reaching consequences [1].

A proposer cannot be alone. A validator that is far from all other validators cannot be a proposer, since it has no viable co-proposers. Every place where proposers are drawn needs both enough stake, because stake implies security, and enough independent validators, so that no single operator controls all proposers of a window. An operator could run several validators to look independent, but every validator pays Alpenglow's validator admission ticket [3], so posing as several means paying it several times. Many validators are also flexible about where they run: most sit in data centers, and moving to a data center near other validators is a modest cost if it makes them eligible as proposers. What matters is the latency between co-proposers, since they exchange their commitments within $2\delta_{\text{local}}$.

### 6.1.1 Hubs

A group of co-located validators is a *hub*. Hubs are not planned: neither the protocol nor a central authority decides where they are. A hub forms wherever enough validators gather and disappears when they leave; the protocol only sets the conditions under which a hub takes part in the proposer schedule, and validators remain free to choose their location [1].

| | |
|---|---|
| **Registration** | Anybody registers a new hub with an on-chain transaction, under a unique, human-readable name such as an airport code (FRA) and with its location as an integer ECEF coordinate triple, as in Validator Location Registration, SIMD-0674 [14]. Registering a name requires a deposit, and the name is released if the hub stays inactive for too long. A new hub is not yet active; it becomes active once enough validators have joined. |
| **Joining** | A validator joins a hub by its name with an on-chain transaction, as part of its registration or later. A validator that joins no hub only votes. |

**Leaving and moving**

Leaving a hub and moving between hubs are on-chain transactions as well. All of them take effect only at an epoch boundary announced in advance, so that schedules are fixed ahead of time, and a cooldown keeps validators from hopping between hubs.

In Acrux, these transactions are the instructions `register_hub`, `join_hub`, `leave_hub`, `move_hub` and `release_hub` of the Hub Registry, a program that runs on Acrux, where its accounts are the schedule's input, and on Solana mainnet. A hub code is 3 to 8 capitals; its location is given as latitude and longitude and stored as integer ECEF metres in WGS84. A membership is pending until the next epoch boundary and then becomes effective, and the member's proposer stake is computed from its stake. The deposit, the registration fee, the cooldown and the inactivity period are set in the registry's configuration. The accounts, instructions and errors are in Chapter 11; an operator reaches all of it from the CLI:

```
acrux hub register FRA --latitude 50.0379 --longitude 8.5622
\n    --identity key.json
acrux hub join FRA --identity key.json
acrux hub move FRA NYC --identity key.json
```

## 6.2 Proposer stake

The paper writes stake as $\rho_i$; this book writes $\sigma_i$ (notation in Chapter 2).

Let $\sigma_i$ be the stake of validator $i$ as a fraction of the total stake. Its *proposer stake* $\sigma'_i$ decides how often it is drawn as a proposer. A validator that is not a member of an active hub has $\sigma'_i = 0$ and only votes. For a member of a hub, the proposer stake is usually just its stake, $\sigma'_i = \sigma_i$, unless the member is too large for its hub: every window needs $k$ distinct proposers, and a single member cannot be more than one of them, however much stake it holds [1].

Let $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_n$ be the stakes of the members of hub $h$, with $n \geq k$. For $j = 0, 1, \ldots, k-1$, let

$$c_j = (1 / (k - j)) \cdot \Sigma_{i>j}\, \sigma_i$$

be the stake of all but the $j$ largest members, divided over the remaining $k - j$ roles, and let the *cap* $c = c_j$ for the smallest $j$ with $\sigma_{j+1} \leq c_j$. Such a $j$ always exists, at the latest $j = k - 1$, where $c_{k-1}$ is the whole remaining stake. The proposer stake of member $i$ is then $\sigma'_i = \min(\sigma_i, c)$, and the *hub stake* is

$$\sigma'_h = \Sigma_{i \in h}\, \sigma'_i = k \cdot c.$$

The $j$ largest members are thus capped at exactly $1/k$ of the hub stake, and they are proposers in every window of the hub, while the other $k - j$ proposers of every window are drawn among the remaining members, in proportion to their stake. The excess stake of a capped member still counts for voting and validator rewards, but not for proposing.

Table 6.1 The paper's example of the cap, $k = 3$, stakes as shares of the total stake. Acrux's fixture `paper-caps.json` holds the same hub in integer stake units and checks `cap_stakes` against it.

| Step | Remaining stake | Cap candidate | Largest remaining member | Result |
|---|---|---|---|---|
| $j = 0$ | 6 + 2 + 1.2 + 0.8 = 10% | $c_0$ = 10% / 3 = 3.33% | 6% > 3.33% | cap the largest, go on |
| $j = 1$ | 2 + 1.2 + 0.8 = 4% | $c_1$ = 4% / 2 = 2% | 2% ≤ 2% | $c$ = 2% |

With $c = 2\%$, the proposer stakes are 2%, 2%, 1.2% and 0.8%, and the hub stake is $6\% = 3 \cdot 2\%$. The two largest members are proposers in every window of the hub. The third proposer is one of the two small members, in 60% and 40% of the windows respectively.

> **A large validator gains nothing from concentrating its stake in one hub, and gains proposer roles by splitting it across hubs in different regions.**

In Acrux, `cap_stakes` (`backend/node/crates/acrux-schedule/src/lib.rs`) computes the cap in integer stake units. It sorts the members by stake, breaking ties by identity, rejects a hub with duplicate members or fewer than $k = 3$ of them, and for $j = 0, 1, 2$ computes $c_j$ as the integer quotient of the remaining stake by $3 - j$, stopping at the first $j$ whose largest remaining member does not exceed it. Each member is returned with its stake, its proposer stake and whether it is capped.

## 6.3 Active hubs

A hub $h$ is *active*, and takes part in the schedule, if [1]

Table 6.2 The conditions for an active hub, and the values Acrux sets where the paper leaves them open.

| | Condition | In Acrux |
|---|---|---|
| (i) | It has at least $n_{\min} \geq k$ members | $n_{\min} = 4$ (`MINIMUM_MEMBERS`) |
| (ii) | Its hub stake is large enough, $\sigma_h \geq \rho_{\min}$ | $\rho_{\min}$ = 1% of the total stake |
| (iii) | Its members reach each other within $\delta_{\text{local}}$ | Enforced by the attestations |

Condition (iii) needs no registry check, and it is individually verifiable: a fast hash exchange proves to one validator that the other one is close, since no message travels faster than light. A member that cannot keep up is marked late by its co-proposers and loses its blocks, so it has every incentive to move to a hub it can actually reach [1]. Acrux sets $n_{\min} = 4$, one more than $k$,

so that an active hub always has a choice among its members; `build_schedule` admits a hub only if it has at least four members and its capped stake is at least 1% of the total validator stake.

The hub parameters $\delta_{\text{local}}$, $\rho_{\text{min}}$ and $n_{\text{min}}$ start permissively, to include as many validators as possible, and can be tightened over time [1]. Since every active hub receives a fixed share of the windows however little stake it holds (Section 6.4), $\rho_{\text{min}}$ and $n_{\text{min}}$ are also what keeps that share from being cheap to obtain.

### 6.3.1 Where Solana's stake sits

The paper measures how much of today's stake is already in a place that could form an active hub. A realistic start for today's validators is a generous $\delta_{\text{local}}$ of 25 ms; with $\rho_{\text{min}}$ = 1%, only 1.2% of the stake would then not yet be in a hub. A smaller $\delta_{\text{local}}$ and a larger $\rho_{\text{min}}$ raise the economic security of the protocol and ask more stake to move [1]. Table 6.3 is the paper's analysis in full.

**Table 6.3** Share of today's stake that could not yet be part of an active hub, if a hub needs at least two validators that reach each other within $\delta_{\text{local}}$ and together hold at least $\rho_{\text{min}}$ of the stake. $\delta_{\text{local}}$ is approximated by distance, with a generous margin over the roughly 1 ms per 100 km of fiber. Based on the 30 largest validator locations, 95.9% of the stake (IP geolocation, validators.solutions, 2026-10-30). From the paper's Table 2 [1].

| $\rho_{\text{min}}$ | ≈ 2 ms (100 km) | ≈ 5 ms (300 km) | ≈ 10 ms (500 km) | ≈ 25 ms (1,000 km) |
|---|---|---|---|---|
| 0.1% | 1.9% | 1.9% | 1.9% | 0.3% |
| 0.5% | 2.7% | 2.7% | 2.7% | 0.7% |
| 1% | 3.7% | 3.7% | 3.7% | **1.2%** |
| 2% | 6.6% | 6.6% | 6.6% | 1.2% |
| 3% | 11.0% | 11.0% | 6.6% | 4.3% |
| 5% | 28.2% | 28.2% | 10.4% | 8.8% |
| 7.5% | 33.5% | 33.5% | 22.5% | 22.0% |
| 10% | 41.3% | 41.3% | 31.2% | 26.7% |

The floor of 1.9% at short distances comes from a few isolated sites with no other validator nearby, independently of $\rho_{\text{min}}$. Since hubs grow bottom-up, the table describes a starting point: wherever enough validators gather, a new hub can form.

### 6.3.2 Candidate hubs

Even though hubs grow bottom-up, the paper asks where they might emerge, and ranks candidate hubs by four criteria: importance for traditional finance, importance for crypto, today's stake, and geographic diversity. Its list suggests starting with three or four hubs, in New York, Frankfurt, Tokyo and London [1]. Acrux Net's genesis hubs are the first three.

Table 6.4 Candidate hubs in a possible order of growth, from the paper's Table 3; bold marks the main reason for a hub's position [1]. TradFi is the rank in the Global Financial Centres Index (GFCI 39, Z/Yen, March 2026); Crypto starts with the rank in the Crypto Friendly Cities Index 2026 (Multipolitan) where available. Stake is the share of today's stake closest to the hub (validators.solutions, 2026-10-30); London includes Amsterdam. Diversity is the distance to the closest hub higher up in the list.

| # | Hub | TradFi | Crypto | Stake | Diversity |
|---|---|---|---|---|---|
| 1 | New York | **#1** | US institutions, ETFs, Coinbase, Gemini | 9% | |
| 2 | Frankfurt | #15; ECB, Eurex | EU under MiCA | **46%** | 6,200 km New York |
| 3 | Tokyo | #10 | **Exchange engines**, Binance, Hyperliquid | 5% | 9,300 km Frankfurt |
| 4 | London | **#2** | Institutional desks, options, Deribit | 27% | 640 km Frankfurt |
| 5 | Singapore | #4 | #1; **Major Asian hub**, trading firms, Bybit | 4% | 5,300 km Tokyo |
| 6 | Hong Kong | **#3**; Shenzhen #9 | #3; Licensed exchanges, gateway to China, OKX | 0.7% | 2,600 km Singapore |
| 7 | San Francisco | **#5** | Crypto companies, venture capital | 4% | 4,100 km New York |
| 8 | Dubai | #7 | #4; **Fast-growing hub**, many exchanges | 0% | 4,800 km Frankfurt |
| 9 | Seoul | #8 | #8; **Large retail volume**, Upbit, Bithumb | 0.3% | 1,150 km Tokyo |
| 10 | Chicago | #14, **derivatives** | Crypto futures, trading firms, CME | 2% | 1,150 km New York |
| 11 | São Paulo | #98; largest in Latin America | Strong adoption in Latin America | 0.4% | **7,700 km** New York |
| 12 | Mumbai | #52; India's financial capital | **High adoption**, restrictive regulation | 0% | 1,900 km Dubai |
| 13 | Sydney | #24; largest in Oceania | Moderate | 0% | **6,300 km** Singapore |
| 14 | Johannesburg | #80; largest in Africa | High adoption in Africa, mostly retail | 0% | **6,400 km** Dubai |
| 15 | Zurich | #11 | #2; **Crypto Valley**, Alpenglow HQ | 0.5% | 300 km Frankfurt |

## 6.4 From hubs to windows

Every window of an epoch is assigned to an active hub, and its $k$ proposers are drawn among that hub's members by proposer stake. If windows went to hubs in proportion to hub stake alone, one large hub could dominate the schedule. Instead, a window is assigned in proportion to hub stake only with probability $\alpha$, and to an active hub chosen uniformly at random otherwise [1]. Acrux uses the paper's $\alpha = 1/2$. A hub holding a fraction $x$ of the total hub stake thus receives a fraction

$$\omega_h = \alpha \cdot x + (1 - \alpha) / |H|$$

of all windows, where $H$ is the set of active hubs. The symbol $\omega_h$ is the paper's. Stake in a large hub therefore earns fewer proposer roles than stake in a small hub, which keeps any single hub from dominating the schedule and gives validators a reason to join smaller hubs.

### 6.4.1 Regional runs

The paper orders the drawn windows with SIMD-0675, the geo-aware leader schedule, with hubs in the role of the leaders [12]: a run of consecutive windows stays within one region, typically moving between nearby hubs, before the schedule moves on to another region. Handovers within a run cost a few milliseconds, and handovers between regions are the same as between distant leaders in Alpenglow. For users, a distant hub is no different from a distant leader in Alpenglow; on average New Constellation is not slower, and users near any hub regularly have their proposers nearby [1]. On Acrux Net, the window handover has a p50 of 149.1 ms from FRA to FRA and 184.3 ms from FRA to NYC (Chapter 9).

acrux-schedule realizes line 5 as a grouping by region: `build_schedule` takes the drawn windows with a stable sort by region, so each region forms one contiguous run with the seeded draw order kept inside it, and records each run as a `RegionalRun` with its region and its first and last window. Each hub carries its region in the schedule input; Acrux Net's three hubs are in three regions, Europe, North America and Asia Pacific, so each run is one hub's windows.

### 6.4.2 The proposers of a window

The hub schedule fixes the hub of every window; it remains to draw the $k$ proposers of each window among the members of its hub $h$. For every window, the members of $h$ are placed in random order on a line of length $\hat{\sigma}_h = k \cdot c$, where member $i$ covers an interval of length $\hat{\sigma}_i$. With a uniformly random offset $u \in [0, c)$, the proposers are the members whose intervals contain the $k$ points $u, u + c, \ldots, u + (k - 1)c$, and their proposer indices are assigned at random [1].

Since no interval is longer than $c$, no two of the $k$ points, which lie exactly $c$ apart, fall into the same interval, so the $k$ proposers are distinct. Since the offset is uniform over $[0, c)$ and the line has length $k \cdot c$, member $i$ is a proposer with probability exactly $\hat{\sigma}_i / c$: a capped member in every window of its hub, every other member in proportion to its stake. The proposers of a window are thus always $k$ different members of the same active hub, so they are co-located, and no validator holds more than one of the $k$ roles of a window. Proposer roles are proportional to proposer stake within every hub, and only partly across hubs, since a fraction $1 - \alpha$ of the windows is assigned independently of hub stake.

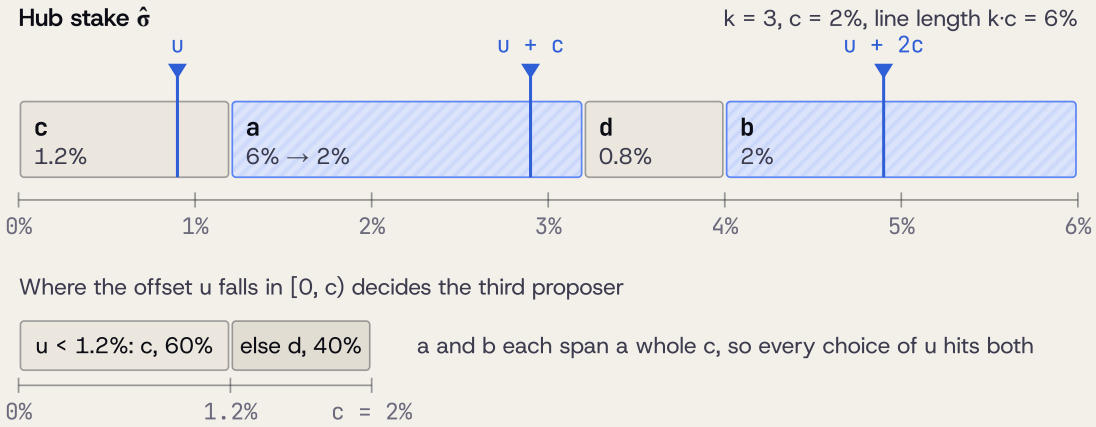



**Figure 6.1** Systematic sampling of $k = 3$ proposers in the paper's example hub, members in one shuffled order. The capped members a and b (blue) each cover a full $c$ and are hit for every offset; the third point falls in c when $u < 1.2\%$ and in d otherwise, which gives them 60% and 40% of the windows.

All draws use a deterministic PRNG seeded from the epoch, and stakes and hub memberships are taken from the same snapshot as the leader schedule, so that all validators compute the same schedule. Algorithm 3 gives the complete procedure.

**Algorithm 3** **NewCon proposer schedule of an epoch with N windows**

**Require:** active hubs $H$; proposer stakes $\hat{\sigma_i}$ and hub stakes $\hat{\sigma_h}$; PRNG seeded from the epoch

```
1  ω_h ← α · σ̂_h / Σ_{g∈H} σ̂_g + (1 - α) / |H| for all h ∈ H
2  for i = 1, …, N do   ▷ hub schedule
3    h_i ← random hub from H, where hub h has probability ω_h
4  end for
5  (h_1, …, h_N) ← SIMD-0675(h_1, …, h_N)   ▷ reorder, with hubs as leaders
6  for i = 1, …, N do   ▷ proposers of window i
7    (m_1, …, m_n) ← shuffle(members of h_i); c ← σ̂_{h_i} / k
8    u ← uniformly random in [0, c); s ← 0; W_i ← ( )
9    for j = 1, …, n do   ▷ m_j covers [s, s + σ̂_{m_j}), the next point is u
10     s ← s + σ̂_{m_j}
11     if u < s then append m_j to W_i; u ← u + c
12   end for
13   W_i ← shuffle(W_i)   ▷ proposer indices
14 end for
15 return (W_1, …, W_N)
```

In acrux-node: `build_schedule` and `cap_stakes`, `backend/node/crates/acrux-schedule/src/lib.rs`; line 5 as the grouping by region of Section 6.4.1.

Lines 1 to 4 draw the hub of each window independently with the probabilities $\omega_h$. Line 5 orders the windows into regional runs by SIMD-0675. Lines 7 to 12 are the systematic sampling of Figure 6.1: walking the shuffled members, the running sum $s$ is the end of the current member's interval, and whenever the next sample point $u$ lies before it, the member is appended and the point moves on by $c$. Since each interval is at most $c$ long, a member is

appended at most once. Line 13 assigns the proposer indices 1 to $k$ at random, which decides, among other things, whose block names the parent on a tie (Chapter 3).

### 6.4.3 The draw in acrux-node

`build_schedule(epoch, seed, hubs, total_stake, windows)` is Algorithm 3 with integer stake. It is deterministic in its inputs:

- **Active hubs.** Hubs with fewer than four members are left out; the others are capped with `cap_stakes` and kept if their hub stake is at least 1% of the total validator stake. The active hubs are sorted by code, so the input order does not matter.
- **The PRNG.** The seed is SHA-256 over the tag `acrux-algorithm-3-v1`, the epoch number and the seed string, and the generator is ChaCha20 (`ChaCha20Rng`). Every draw of the epoch, the hubs, the shuffles and the offsets, comes from this one stream in a fixed order.
- **Hub of a window.** For each window, a fair coin decides between the two terms of $\omega_h$: on one side a hub is drawn in proportion to hub stake, by an integer drawn uniformly below the total hub stake; on the other a hub is drawn uniformly. This realizes $\omega_h$ with $\alpha = 1/2$ exactly, without floating-point probabilities.
- **Proposers.** The members are shuffled, and the sampling runs on a line scaled by $k$: each member covers $k \cdot \sigma_i$, the offset is an integer below $\sigma_h$, and the points are $\sigma_h$ apart. This is the same sampling with $c = \sigma_h / k$, computed without losing the fractional part of $c$. The chosen three are shuffled for their indices.

The node and the API's schedule preview on `/hubs` compute the schedule through this one code path (`acrux schedule` reads hubs as JSON on standard input), and a shared fixture, `backend/node/fixtures/schedule/genesis.json`, pins the output both must produce. Figure 6.2 draws that fixture: epoch 43, 48 windows, the three genesis hubs, each with members of the paper's example stakes 6, 2, 1.2 and 0.8 units out of 30 in total.

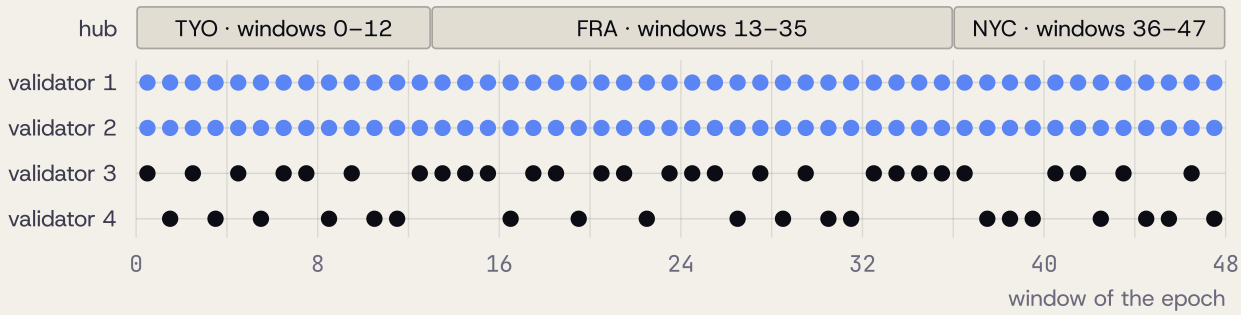
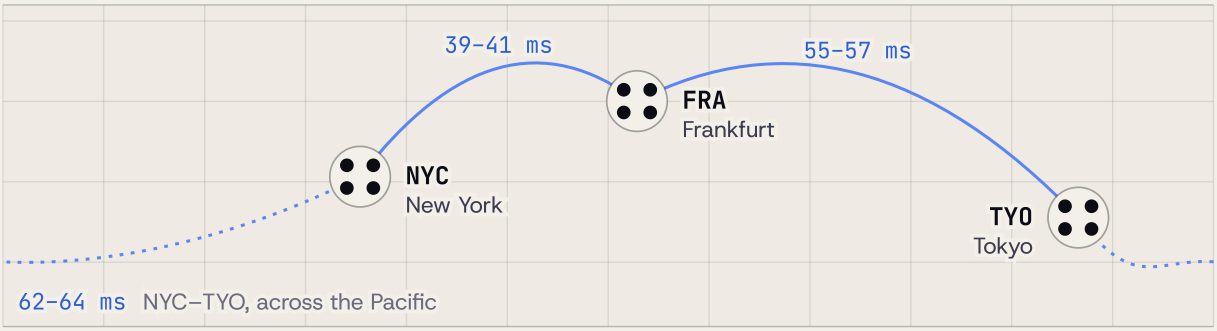



**Figure 6.2** The shared schedule fixture: 48 windows of epoch 43 over FRA, NYC and TYO, each hub's members with stakes in the proportions 6 : 2 : 1.2 : 0.8. Each hub has $\omega_h = 1/3$; the draw gave TYO 13 windows, FRA 23 and NYC 12, grouped into one regional run each. Validators 1 and 2 are capped (blue) and propose in every window of their hub; validators 3 and 4 share the third role, 28 and 20 windows across the three hubs.

## 6.5 Acrux Net's hubs

Acrux Net runs the three genesis hubs FRA, NYC and TYO, with four validators each, twelve in all. Each validator carries the coordinates of its hub's airport, and the transport applies to every link the delay of the distance between the cities (Chapter 8). Figure 6.3 shows the hubs and the one-way delays on the wire.

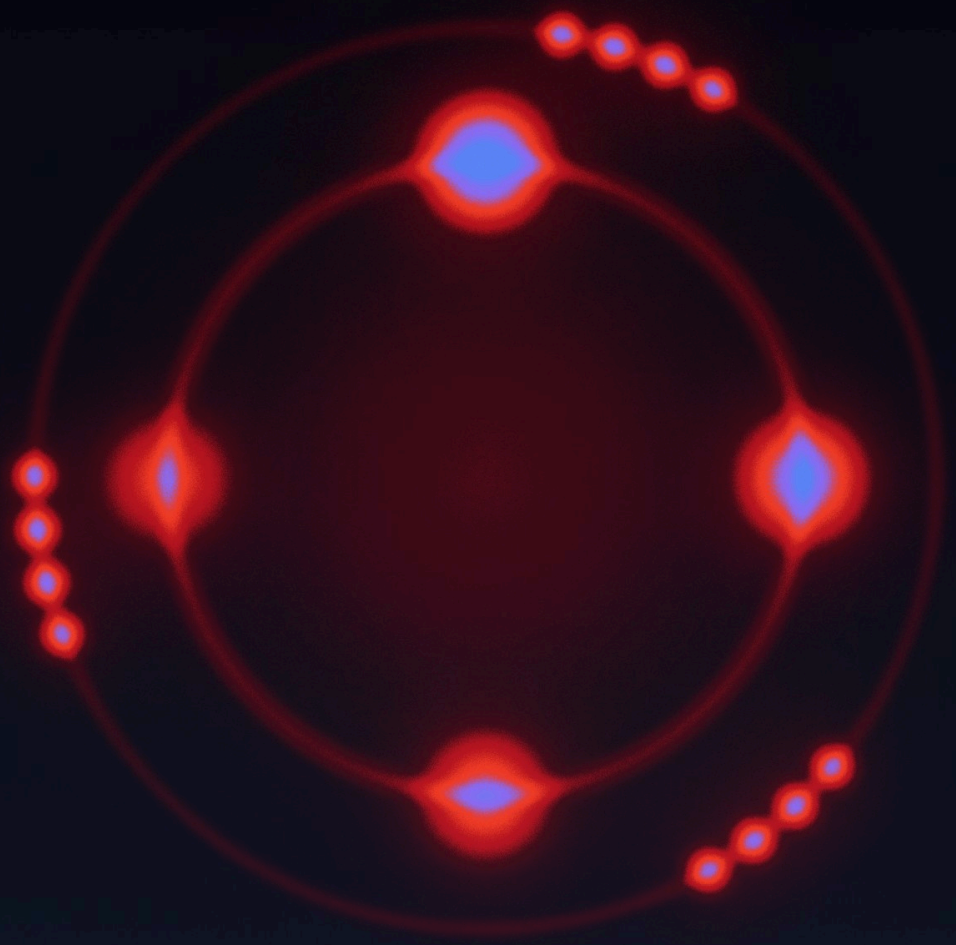



Inside a hub: 1–4 ms between its four validators — One-way delay on the wire, Acrux Net

**Figure 6.3** Acrux Net's genesis hubs by longitude and latitude, four validators each, with the one-way link delays applied on the wire. NYC–TYO runs over the far side of the strip (dotted). Inside a hub, the 1 to 4 ms between validators keep the $2\delta_{\text{local}}$ = 20 ms commitment exchange well within reach.

All twelve validators hold equal stake. Each hub then holds a third of the total, every hub has $\omega_h = 1/3$, and the cap of a hub is 4/3 of a member's stake, so no member is capped: each of the four members of a hub is one of its three proposers in 3/4 of the hub's windows. acrux-node draws Acrux Net's schedule with `build_schedule` as a sequence of 96 windows, 76.8 s at 800 ms per window, and runs through it in turn. The live schedule ahead is on `/net`, and the preview of the next epoch, from the registry and a stake snapshot, on `/hubs`.

Validators that are not among a window's proposers keep validating and voting as in Chapter 4, wherever they are. Joining a hub is what makes a validator a proposer, and the Hub Registry is open for registering and joining hubs now, on Acrux and on Solana mainnet (Chapter 11).



7

# Correctness

Safety and correct state hashes, selective-censorship resistance, proposal hiding and accountable equivocation: the theorem and its three lemmas, proved.

New Constellation treats Alpenglow almost as a black box and proves four properties on top of it. This chapter states each as the paper states it, gives the paper's proof, and names the Fault Lab scenario that runs it on a cluster.

## 7.1 The model

The paper changes two things in Alpenglow [3]: the definition of the block hash, which for a cblock covers its parent, its agreed sets, its attested commitments and its state hash, and the preconditions under which a validator emits its local events, its `Block` event and its timeouts (Chapter 4). Everything else is Alpenglow's, and the proofs rest on five assumptions [1].

| | |
|---|---|
| **(M) Model** | The byzantine stake $f$ among the validators is less than 20%. After GST, at least 60% of the stake is correct and online, and messages between correct nodes take at most $\delta_{\text{global}}$. Messages from a correct proposer reach the co-proposers in at most $\delta_{\text{local}}$. |
| **(D) Determinism** | Merging, transaction ordering and execution are deterministic. |
| **(T) Timing** | The parameters satisfy $\Delta_{\text{timeout}} \geq 3\delta_{\text{global}}$ and $2\delta_{\text{local}} \leq \delta_{\text{global}}$. After GST, a block of a correct proposer reaches all correct validators via Rotor at most $2\delta_{\text{global}}$ after the proposer completed it. A correct validator executes a slot within $\delta_{\text{block}}$, once it holds the blocks and attestations of the slot and the state it builds on. |
| **(AS) Alpenglow safety** | If correct nodes follow the Alpenglow voting rules, the finalized blocks form a single chain. At any point in time the finalized chains of different correct validators are prefixes of one or the other. This holds for every block-validity predicate, regardless of when correct nodes cast their initial votes, and also if correct nodes omit fallback votes. |
| **(AL) Alpenglow liveness** | Consider it is after GST. (i) If every correct validator eventually casts either a notarization or skip vote for every slot, every slot obtains a notar-fallback or skip certificate at every correct node. (ii) If the same valid block in a slot reaches all correct nodes before the timeouts, it is finalized. This is especially the case if the leader is correct. |

Acrux meets (T) with its parameters (Appendix A): $\Delta_{\text{timeout}}$ = 750 ms is exactly $3\delta_{\text{global}}$, the lower bound (T) allows, and $2\delta_{\text{local}}$ = 20 ms is far below $\delta_{\text{global}}$ = 250 ms. (D) holds by construction of the merge and the selection

(Chapter 5) and is checked continuously: validators compare their state hashes, 10,564 comparisons without a mismatch in the 600 s run of Acrux Net (Chapter 9).

The proofs use the vocabulary of Chapter 3 and Chapter 4, whose notation table is in Chapter 2. An attestation of proposer $q$ for slot $s$ carries a vector $\pi$ with $\pi[j]$ the commitment of $p_j$ if it arrived on time and $\perp$ otherwise; the paper writes this vector $\tau$. Two attestations *agree* if their vectors are equal. A slot is *agreed* on a set $P$ at a validator if (a) all attestations for the slot it received agree and attest exactly $P$, (b) it holds the attestation of every proposer in $P$, (c) $|P| > k/2$, and (d) it has seen no equivocation in the slot. A proposer *equivocates* if it signs two different proposer blocks, commitments or attestations for the same slot. A window *starts* when the first correct node emits ParentReady for it, and a *crashed* proposer sends no messages in the window.

## 7.2 Correctness

**Theorem 2.1 · NewCon correctness**

NewCon is safe and every certified state hash is correct. After GST, every window obtains ParentReady. Moreover, a cblock is finalized in every consensus slot of every window that starts after GST, has $> \lfloor k/2 \rfloor$ correct proposers while all its other proposers are crashed, and whose parent every correct validator has executed when it emits ParentReady for the window. Here, a window starts when the first correct node emits ParentReady for it, and a crashed proposer sends no messages in the window.

***Proof.*** **Safety.** View the cblock of consensus slot $i$ as the block of a virtual leader, with hash $h(c_i)$: validators run the Alpenglow voting rules on this block hash. Like a regular block hash in Alpenglow, $h(c_i)$ determines the block and its parent: it covers the hash of the parent cblock, the agreed sets and the attested commitments of all slots of $c_i$, and every commitment covers its proposer block. Per consensus slot, a correct validator emits at most one `Block` event (Algorithm 2, lines 44 and 50), and thus casts at most one notarization vote. Otherwise, only the validity predicate of blocks and the timeouts differ from Alpenglow, and both only decide whether and when a correct validator casts its initial vote. So (AS) gives a single finalized chain of cblocks. Since a commitment binds both the ciphertext and the key of its proposer block, this chain binds uniquely to a set of proposer blocks and their payloads across the corresponding slots.

**State hashes.** With (D), the slots of a chain up to a cblock have a unique post-execution state, the result of executing their merged transactions in order; its hash is the *correct* state hash of the cblock. A correct validator computes $h_{\text{state}}$ for $c_i$ by executing the slots of $c_i$ on the state it computed for the parent of $c_i$ (Algorithm 2, lines 33, 38 and 50), so by induction along the chain, a correct `NotarVote` is for a cblock that contains the correct state

hash. Every notarization and notar–fallback certificate has 60% of the stake and thus contains a correct vote. A correct `NotarFallbackVote` requires seeing at least 20% > $f$ notarization votes for the same hash, and hence a correct `NotarVote`. So every certified cblock contains the correct state hash.

**ParentReady.** By induction over the windows, every correct node eventually emits ParentReady for every window. Suppose that every consensus slot before some window has a notar–fallback or skip certificate at every correct node, which is trivially true for the first window. Take the last of these slots with a notar–fallback certificate, or the genesis block if there is none: all consensus slots after it have skip certificates, so every correct node emits ParentReady for the window. All correct validators then set the timeouts of the window (Algorithm 2, line 6) and cast a notarization or skip vote in each of its consensus slots by these timeouts. By (AL)(i), every consensus slot of the window obtains a notar–fallback or skip certificate at every correct node, which completes the induction.

$\theta_p$ is a time; the paper writes it $\sigma_p$.

**Finalization.** Consider a window as in the theorem, with first consensus slot $i_0$, and let $A$ be the set of its correct proposers. All times are real times, and clock drift within a window is ignored. Let $t_u$ be the time when a correct node $u$ first emits ParentReady for $i_0$; by then, $u$ has broadcast the certificates that triggered it. With (M), $|t_p - t_v| \le \delta_{\text{global}}$ for every proposer $p \in A$ and every correct validator $v$, and $|t_p - t_q| \le \delta_{\text{local}}$ for all $p, q \in A$.

Consider the $m$–th slot of the window. A proposer $p \in A$ completes its first block $\delta_{\text{block}}$ after ParentReady, even if it started early, and then produces its blocks back to back, so it completes its block of this slot and sends its commitment at time $\theta_p = t_p + m\delta_{\text{block}}$ (Algorithm 1, lines 9, 11 and 16). It starts the hash exchange at $\text{start}_p \le \theta_p$, the earlier of $\theta_p$ and the arrival of the first commitment of a co–proposer (Algorithm 1, lines 14 and 21). Let $\theta = \min_{q \in A} \theta_q$, so $\theta_q \le \theta + \delta_{\text{local}}$ for all $q \in A$. Crashed proposers send no commitments, and correct ones send none before $\theta$, so $\text{start}_p \ge \theta$. The commitment of $q \in A$ reaches $p$ by $\theta_q + \delta_{\text{local}} \le \theta + 2\delta_{\text{local}} \le \text{start}_p + 2\delta_{\text{local}}$, that is, on time (Algorithm 1, line 28). So all proposers in $A$ send the same attestation vector: it attests exactly the proposers in $A$, with the commitments of their blocks. Proposer $p$ sends its attestation at the latest at $\text{start}_p + 2\delta_{\text{local}} \le \theta_p + 2\delta_{\text{local}}$ (Algorithm 1, lines 24 and 32).

The attestation of $p$ reaches a correct validator $v$ by $\theta_p + 2\delta_{\text{local}} + \delta_{\text{global}}$, and the block of $p$ by $\theta_p + 2\delta_{\text{global}}$ by (T). As $\theta_p \le t_v + \delta_{\text{global}} + m\delta_{\text{block}}$ and $2\delta_{\text{local}} \le \delta_{\text{global}}$, validator $v$ holds the blocks and the attestations of all proposers in $A$ for the $m$–th slot by time $a_m = t_v + m\delta_{\text{block}} + 3\delta_{\text{global}}$. Correct proposers do not equivocate, and nobody else can sign in the name of a proposer, so $v$ never sees an equivocation or any other attestation for a slot of the window: at the latest from $a_m$ on, the $m$–th slot is agreed on $A$ (Algorithm 2, lines 54, 55 and 57), the same set in every slot (Algorithm 2, line 27). The parent of $c_{i_0}$ is named by some $p \in A$, which emitted ParentReady for it at $t_p$; the certificates of $p$ reach $v$ by $t_p + \delta_{\text{global}} \le a_1$, and $v$ has executed this parent by assumption (executing it by $a_1$ would suffice). By (T), $v$ executes a slot within $\delta_{\text{block}}$ once it holds its blocks, its attestations and the preceding state. By

induction on $m$, it has executed the $m$–th slot by $a_m + \delta_{\text{block}}$: its blocks and attestations are there by $a_m$, and so is the preceding state, as $a_{m-1} + \delta_{\text{block}} = a_m$ (for $m = 1$, this is the state of the parent).

Consensus slot $i$ of the window ends with the $j\kappa$–th slot of the window, where $j = i - i_0 + 1$. So $v$ has executed $c_i$ by

$$a_{j\kappa} + \delta_{\text{block}} = t_v + 3\delta_{\text{global}} + j\delta_{\text{cblock}} + \delta_{\text{block}} \le t_v + T(i),$$

that is, before its timeout, which expires $T(i) = \Delta_{\text{timeout}} + j\delta_{\text{cblock}} + \delta_{\text{block}}$ after ParentReady (Algorithm 2, line 6), with $\delta_{\text{cblock}} = \kappa\delta_{\text{block}}$; the inequality is $\Delta_{\text{timeout}} \ge 3\delta_{\text{global}}$ from (T). If some proposers are crashed, their attestations are missing, and $v$ waits for them before its first vote of the window, but at most until $t_v + \delta_{\text{block}} + 2\delta_{\text{global}} + 2\delta_{\text{local}} \le a_1$ (Algorithm 2, lines 7 and 46); so in any case, $v$ emits `Block` for $c_i$ before its timeout (Algorithm 2, line 50).

By (D), all correct validators obtain the same cblock, including its parent and state hash, and thus the same hash $h(c_i)$. So in every consensus slot of the window, the same valid block reaches all correct nodes before the timeouts, just as with a correct leader, and (AL)(ii) applies. ■

### 7.2.1 The bounds with Acrux's parameters

With $\delta_{\text{block}} = 50$ ms, $\kappa = 4$, $\delta_{\text{local}} = 10$ ms, $\delta_{\text{global}} = 250$ ms and $\Delta_{\text{timeout}} = 750$ ms, every bound of the proof is a number of milliseconds after the validator's own ParentReady. Because Acrux sets $\Delta_{\text{timeout}}$ exactly to $3\delta_{\text{global}}$, the bound on execution meets the timeout with equality in every consensus slot of the window.

**Table 7.1** The time bounds of the finalization proof on Acrux, in ms after $t_v$, the validator's ParentReady for the window.

| Bound | Formula | Acrux |
|---|---|---|
| Attestation deadline (Algorithm 2, line 7) | $\delta_{\text{block}} + 2\delta_{\text{global}} + 2\delta_{\text{local}}$ | 570 |
| First slot's blocks and attestations held, $a_1$ | $\delta_{\text{block}} + 3\delta_{\text{global}}$ | 800 |
| $m$–th slot executed | $a_m + \delta_{\text{block}} = (m+1)\delta_{\text{block}} + 3\delta_{\text{global}}$ | $750 + 50(m+1)$ |
| $c_{i_0}$ executed, and $T(i_0)$ | $j = 1$ | 1,000 |
| $c_{i_0+1}$ executed, and its timeout | $j = 2$ | 1,200 |
| $c_{i_0+2}$ executed, and its timeout | $j = 3$ | 1,400 |
| $c_{i_0+3}$ executed, and its timeout | $j = 4$ | 1,600 |

The attestation deadline, 570 ms, falls before $a_1$, 800 ms, so the wait for missing attestations never delays the first vote past the point the proof needs. These are worst-case bounds under (M) and (T). On Acrux Net, where links run far below $\delta_{\text{global}}$, a cblock finalizes at a p50 of 324 ms (Chapter 9).

### 7.2.2 What each condition covers

The conditions on the crashed proposers and on the parent are both part of the statement of finalization. A proposer that crashes within a window, rather than from its start, may reach only some nodes with its last commitment or attestation, so that the attestations disagree; the theorem covers proposers that are crashed for the whole window, and a crash in the middle of a window is covered by safety, as a skip. And since validators execute before they vote, a validator that has not yet executed the parent of a window, for example because it has to repair it first, is outside the timing argument for that window. If (D) failed, for instance through an implementation bug, correct validators would compute different state hashes for the same proposer blocks and thus different cblock hashes, which shows in the voting phase or when a validator learns the consensus outcome [1]; Acrux's state-hash comparisons are that check, run on every slot.

**In the Fault Lab.** The *Crash* scenario takes proposer 3 of window 4 down from the window's start (Chapter 10). Its co-proposers attest it as $\perp$, their attestations agree, and $P = \{1, 2\}$ is a strict majority of the three: every slot of the window is agreed on two blocks and executes without delay. Validators hold their vote for the window's first cblock until the attestation deadline, 570 ms after ParentReady, and then vote for the rest of the window as each cblock executes; the window finalizes on two proposers. In acrux-node, the Alpenglow side of the theorem is `Votor::drive` and `Votor::ready_parent` (`backend/node/crates/acrux-consensus/src/lib.rs`), and the 600 s run of Acrux Net finalized 2,640 cblocks with no skip.

## 7.3 Selective-censorship resistance

Selective-censorship resistance is the property that Garimidi, Neu and Resnick set for multiple concurrent proposers [5]: the proposers that want a transaction out cannot keep the block of an honest proposer that carries it out of the slot. In New Constellation it holds per slot.

**Lemma 2.2 · Selective-censorship resistance**

In a window that starts after GST, for every slot, either the proposer blocks of all correct proposers are part of the slot or the corresponding consensus slot is skipped.

***Proof.*** Let $p$ be a correct proposer of the window, $c_{i_0}$ the first cblock of the window, and $v$ a correct validator, with the notation of the proof of Theorem 2.1. Proposer $p$ sends its attestation for a slot only after its own commitment, so the attestation attests its own block (Algorithm 1, lines 24 and 28), and at most $2\delta_{\text{local}}$ after completing this block, since start[$s$] is never later than that (Algorithm 1, lines 14 and 24). It completes its block of the first slot of the window at $t_p + \delta_{\text{block}}$, so its attestation for this slot reaches $v$ by $t_p + \delta_{\text{block}} + 2\delta_{\text{local}} + \delta_{\text{global}} \le t_v + \delta_{\text{block}} + 2\delta_{\text{global}} + 2\delta_{\text{local}}$, that is, by the deadline of $v$ for missing attestations (Algorithm 2, line 7).

Now suppose that $v$ emits `Block` for a cblock $c_i$ of the window whose slots are agreed on a set $P$ that does not contain $p$. Then $v$ has emitted `Block` for $c_{i_0}$ before (Algorithm 2, line 47), and all slots of the window up to $c_i$ are agreed on the same set $P$ (Algorithm 2, line 27), in particular the first slot of the window. So when $v$ emitted `Block` for $c_{i_0}$, all attestations for this first slot that it had received attested exactly $P$ (Algorithm 2, lines 44 and 55), and that of $p$ was not among them. But at that time, $v$ either held the attestations of all proposers for the first slot, or its deadline had passed (Algorithm 2, line 46); in both cases it held the attestation of $p$, a contradiction.

So no correct validator casts a notarization vote for a cblock of the window in which the block of a correct proposer is missing. As in the proof of Theorem 2.1, such a cblock obtains neither a notarization nor a notar-fallback certificate, so its consensus slot can only be skipped. ■

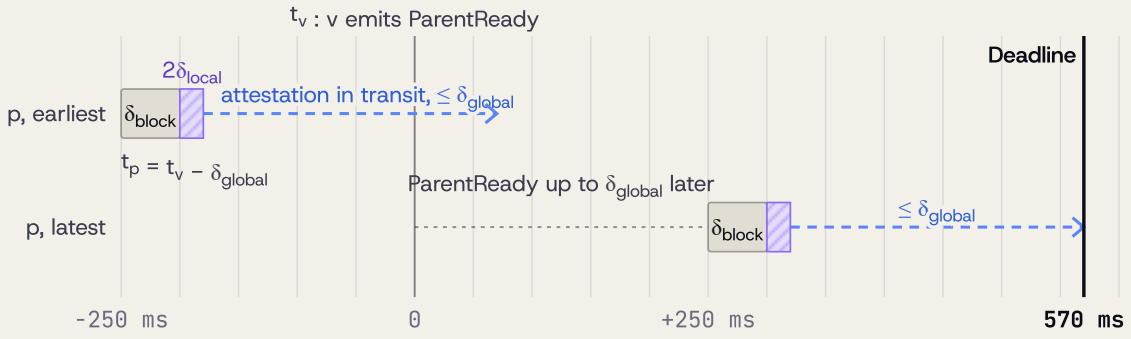



**Figure 7.1**

Why the deadline suffices, on Acrux's parameters. A correct proposer $p$ emits ParentReady at most $\delta_{global}$ after the validator $v$; it completes its first block $\delta_{block}$ later, attests within $2\delta_{local}$ (violet), and its attestation travels at most $\delta_{global}$. In the latest case it lands exactly at $v$'s deadline, 570 ms after $t_v$.

> **Colluding proposers cannot exclude a correct co-proposer by attesting only each other: the validator holds its first vote of the window until that co-proposer's attestation must have arrived.**

For a user this is the guarantee that a transaction needs one non-censoring proposer among those it was sent to: unless the slot is skipped, the block of that proposer is in the slot, and the transaction with it. The wait is needed once per window, since the agreed set cannot change after the first slot; in a window with all proposers correct, every attestation arrives well before the deadline and nothing waits.

**In the Fault Lab.** The *Censorship* scenario sends a signed transfer to all three proposers of window 4 for slot 66; proposers 1 and 2 leave it out, proposer 3 includes it. Proposer 3's attestation for the window's first slot reaches every correct validator by the deadline, the agreed set is $P = \{1, 2, 3\}$, and the transaction executes in slot 66 from proposer 3's block and finalizes. The *Split commitments* scenario shows the other branch of the lemma: proposer 3 sends its commitment to co-proposer 1 only, the attestation vectors differ, no validator considers the slot agreed, and the cblock is skip-certified

(Chapter 10). In acrux-node the lemma lives in `Network::try_execute` and `Network::try_vote` (`backend/node/crates/acrux-node/src/protocol.rs`) and the agreement rule in `agreed` (`agreement.rs`).

## 7.4 Proposal hiding

The last look is the advantage of a block producer that sees the others' transactions before it fixes its own. Commit-and-reveal removes it for blocks: every block is disseminated encrypted under a fresh key, the commitment binds both the ciphertext and the key, and the key is released only in the attestation, after the hash exchange (Chapter 3).

**Lemma 2.3 · Proposal hiding**

It is impossible for a malicious proposer to craft a proposer block based on any information within the payload of a correct proposer's proposer block and have the attacker and victim block appear in the same cblock.

***Proof.*** Until a proposer receives the attestation containing the decryption key, it holds at most the full encrypted proposer block of a specific other proposer (Algorithm 1, line 8), and thus learns nothing about its payload. Further, the key is only sent in the attestation (Algorithm 1, line 32), which contains the commitments of the other proposers, including the attacker, to their proposer blocks. So the attestation of the victim proposer includes either (a) a positive attestation about the attacker proposer, of a commitment for a block that was created without knowing the payload of the victim block, or (b) a negative attestation about the attacker proposer. By the binding property of the commitment, and since the commitment binds both the ciphertext and the key, the attacker cannot later open the commitment to a different block, in particular not to one chosen after the commitment was sent.

If the victim proposer block is included in the cblock that a correct validator voted notarize for, the attestations of all accepted proposers agree with the victim's (Algorithm 2, line 55), so that same validator's notarization vote covers, depending on the case above, (a) a block of the attacker proposer which was crafted without knowing the contents of the victim block, or (b) no block of the attacker proposer. This holds for all correct validators' notarization votes, thus also for the notarization or notar-fallback certificate, and thus for the finalized cblock. ■

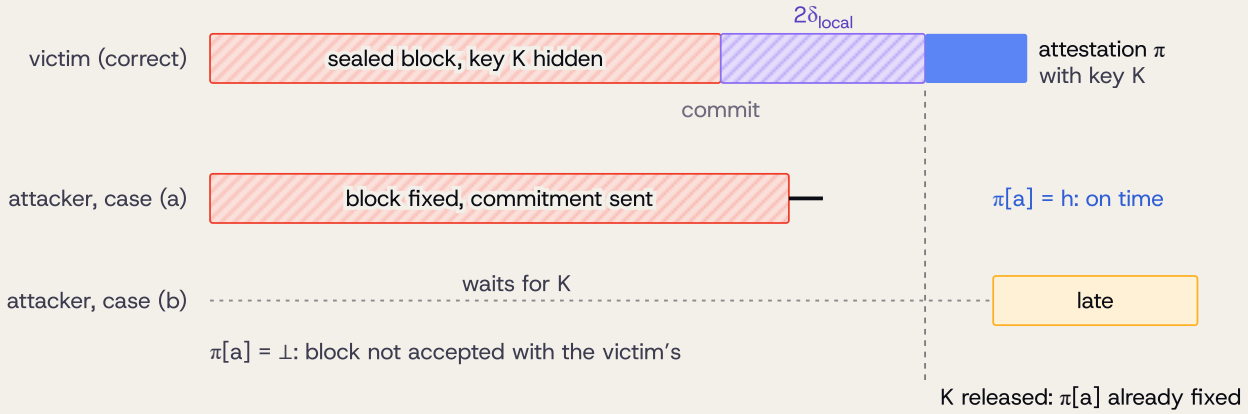



**Figure 7.2** The two cases of the proof. The victim's key leaves only in its attestation $\pi$, at most $2\delta_{local}$ after its commitment. Either the attacker's commitment was in by then, and its block was fixed before the key was out (a), or the victim attests it $\perp$, and a block the attacker builds after learning the key can never be accepted together with the victim's (b).

In Acrux, the commitment is the hash of the block as disseminated, with the hash of the key in the header, so it binds both; the block cipher is ChaCha20-Poly1305 under a fresh key per block (`encrypt_slice` and `decrypt_slice`, `backend/node/crates/acrux-crypto/src/lib.rs`; constructions in Chapter 8). Proposal hiding covers whole blocks. Hiding a single transaction from the proposers that receive it is the work of threshold transactions (Section 5.3), and the two compose: a 2-of-3 transaction is unreadable to each of its receivers, and the slot it lands in is fixed before any of its blocks is readable.

**In the Fault Lab.** The *Last look* scenario has proposer 3 of window 4 send its commitment 28 ms late, past the 20 ms exchange window, to see the others first. Its co-proposers close their exchange without it, attest it $\perp$ and release their keys: case (b). Proposer 3's own attestation names all three commitments, so the vectors disagree, the late block is dropped and the cblock is skip-certified. The *Private front-run* scenario runs a 2-of-3 transfer against a receiver that tries to read it: it holds one of the two fragments it would need, rebuilds neither the ciphertext nor the key, and the transaction executes in its slot (Chapter 10).

## 7.5 Accountable equivocation

A cblock is built by several proposers, so an equivocation on it could in principle arise without any single signer at fault. The strict-majority rule of agreement rules this out: an effective equivocation on a cblock can always be blamed on a misbehaving proposer [1].

**Lemma 2.4 · Accountable equivocation**

If two correct nodes vote notarize for a different cblock, there exists proof of equivocation blaming at least one of the proposers between the two correct nodes. A correct proposer is never falsely blamed.

***Proof.*** For the correct nodes to vote notarize on different cblocks $c_i \neq c'_i$, there must be xblocks $x_s \neq x'_s$ such that one correct node saw unanimous attestations of $> \lfloor k/2 \rfloor$ proposers supporting the proposer blocks making up $x_s$ (Algorithm 2, lines 55 and 57), and similarly for the other block and the other node. Two sets of more than $\lfloor k/2 \rfloor$ of the same $k$ proposers intersect, so there is at least one proposer who signed both conflicting attestations, over different sets of proposer blocks for the same slot. This proposer can be blamed as misbehaving. A correct proposer never signs two different attestations for the same slot (Algorithm 1, line 24), and is thus never blamed by such a proof. ■

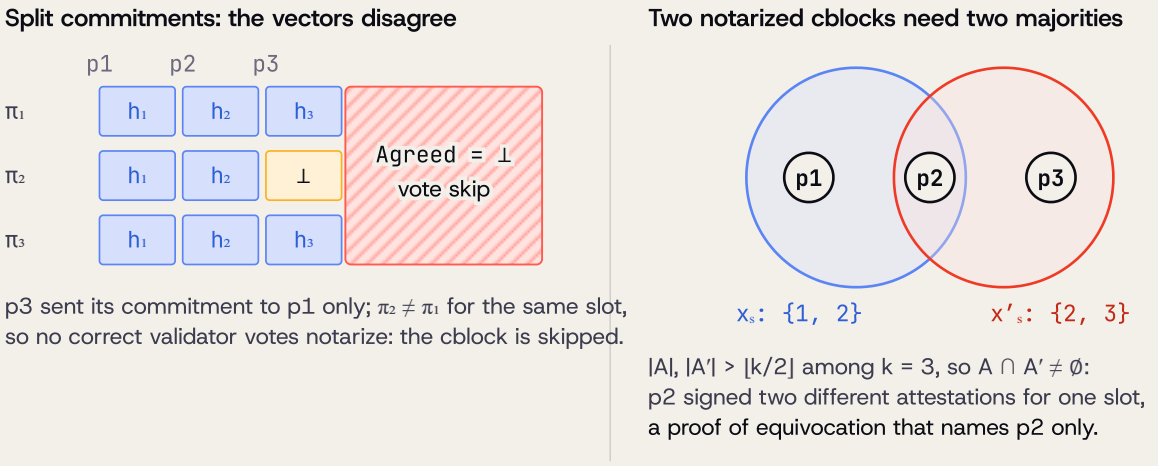



**Figure 7.3**

Left: disagreeing vectors for one slot are not agreed, so validators vote skip and no two of them notarize different cblocks. Right: for two correct nodes to notarize different xblocks of slot $s$, each needs unanimous attestations from a strict majority of the $k = 3$ proposers; any two such majorities share a proposer, whose two signed attestations are the proof.

The lemma holds only because the threshold for the attestations is $> \lfloor k/2 \rfloor$. The paper shows it with a protocol changed to let $\lfloor k/2 \rfloor$ attestations decide the block content [1]:

**Example 2.5.** *Consider a NewCon execution with $k = 2$ proposers. The local network has an intermittent problem, causing both proposers to miss the other's proposal commitment before the deadline, so they each include only their own proposal in their attestation. Further, the global network behaves such that some correct validators see only the one attestation and some only the other, each are considered valid and either set of validators votes on a different cblock made up of only one of the proposer blocks each. This is effectively an equivocation in the Alpenglow protocol. Because this happened even though both proposers were honest, neither proposer can clearly be blamed and no strong penalty, like slashing, can be applied.*

With the strict majority, the same network fault gives two single-proposer sets, neither of which is a majority of two, so no slot is agreed and the consensus slot is skipped. This is also why $k = 3$ is the smallest number of proposers that tolerates one crashed proposer: the two that remain are still a strict majority.

**In the Fault Lab.** The *Equivocation* scenario has proposer 3 send its co-proposers a second commitment for slot 64 with a different hash, signed with the same key. The two signed commitments are a slashing proof: anyone can check both signatures against the proposer's identity, and the proof names proposer 3 and no one else. A validator that has seen it cannot consider the slot agreed, by condition (d), so the cblock is skip-certified (Chapter 10). In acrux-node, `ExchangeEngine::receive_commit` (`exchange.rs`) turns two signed commitments into a `Wire::Equivocation` proof, and `Network::receive` marks the slot; the proof's format is in Appendix D.

## 7.6 What the design gives

New Constellation requires unanimous attestations from a majority of the proposers and votes skip if any disagree. This choice maximizes censorship resistance while tolerating crashed but otherwise honest proposers [1]. The paper places it against the other end of the range: the proposers of a window could run a local synchronous one-shot BFT protocol to agree on the set of proposer blocks that make up the cblock, which would let a super-majority of more than 2/3 always propose a block, in the face of a minority of not just crashed but also byzantine proposers, and would give any super-majority coalition of proposers absolute power of selective censorship. New Constellation keeps that power from every coalition: a byzantine proposer can make its slot skip, and it cannot make a correct proposer's block disappear from a slot that executes.

> **The guarantees are paid for with skips, never with a wrong block: every certified state hash is correct, and every disagreement ends in a skip certificate.**

**Table 7.2** Each property, where the paper proves it, and the Fault Lab scenario that shows it on a sandbox of twelve validators (Chapter 10).

| Property | Paper | Scenario | Outcome recorded |
|---|---|---|---|
| Crash tolerance | Theorem 2.1 | Crash | The window continues with $P = \{1, 2\}$; the first vote waits for the attestation deadline |
| Censorship resistance | Lemma 2.2 | Censorship | The honest proposer's transaction enters the agreed xblock and finalizes |
| Skip on disagreement | Lemma 2.2 | Split commitments | Different attestation vectors prevent agreement and yield a skip certificate |
| No last look | Lemma 2.3 | Last look | The late proposal is dropped and disagreeing vectors produce a skip certificate |
| Accountable equivocation | Lemma 2.4 | Equivocation | The two signed commitments form a slashing proof and the cblock is skipped |
| Transaction hiding | Section 3.2; Lemma 2.3 | Private front-run | One fragment is unreadable; the honest same-slot fragments reconstruct the swap after commitment |

6 CRATES · acrux CLI
ED25519 · CHACHA20-POLY1305 · REED-SOLOMON · SHAMIR
LITESVM EXECUTION



8

# The implementation

acrux-node is a Rust workspace of six crates: the four algorithms, Rotor and Votor on cblocks, SVM execution, QUIC with each link's geography, a store, a gateway and the acrux CLI.

acrux-node is the first implementation of New Constellation. It is one Rust workspace and one binary, `acrux`, that runs a whole net: twelve validators with their own clocks, keys, SVMs and Votor instances, the QUIC links between them, a store, a gateway and the Fault Lab. This chapter reads it from the code.

## 8.1 The workspace

The workspace lives in `backend/node/`. Its `Cargo.toml` lists six member crates under `crates/`, all at version 0.2.0 on the 2021 edition, and pins every dependency once at the workspace level. Five crates are libraries with one concern each; the sixth, `acrux-node`, holds Algorithms 1 and 2, every surface the net exposes, and the binary. The two Solana dependencies are pinned exactly, `litesvm = 0.6.1` and `solana-sdk = 2.2.1`, so the SVM every validator runs is one version.

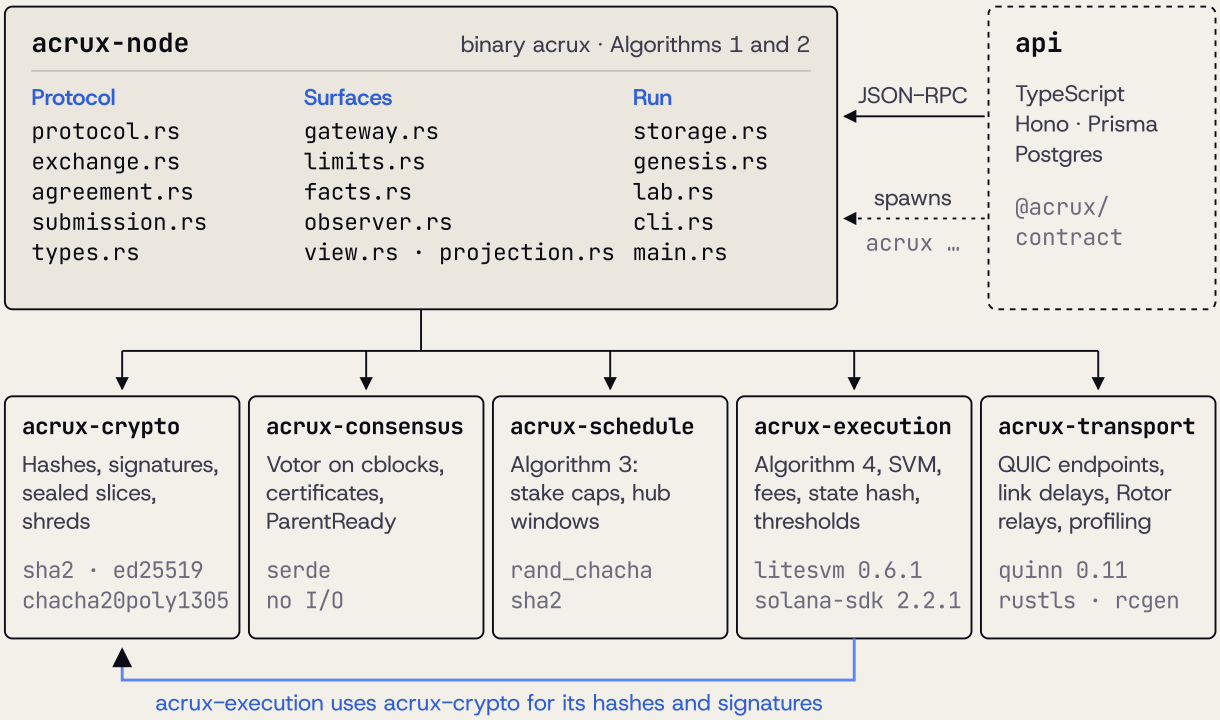



Figure 8.1 The workspace. `acrux-node` depends on all five library crates; among the libraries only `acrux-execution` depends on another. The API calls the gateway over JSON-RPC and runs the same binary as a child process for the schedule preview and the Fault Lab.

Each library compiles and tests without the others, and `acrux-node/src/lib.rs` re-exports them under the module paths the node uses. Table 8.1 gives each crate's concern and the functions to read first.

Table 8.1 The six crates, what each owns, and where to start reading.

| Crate | Owns | Start with |
|---|---|---|
| `acrux-crypto` | SHA-256, Ed25519 with strict verification, ChaCha20-Poly1305 slices with H(K) bound in the commitment, Reed-Solomon shreds 64/32 with Merkle proofs | `encrypt_slice, encode_slice, verify_shred, reconstruct_verified_slice` |
| `acrux-consensus` | Votor on cblocks: notarization 60%, notar-fallback, skip, fast finalization 80%, finalization 60%, certificates, ParentReady, safe-to-notar and safe-to-skip | `Votor::local_block, timeout, receive, drive, ready_parent` |
| `acrux-schedule` | Algorithm 3: proposer-stake caps, $\alpha = 1/2$, regional runs, systematic sampling, an epoch-seeded ChaCha20 draw | `cap_stakes, build_schedule` |
| `acrux-execution` | Algorithm 4, execution on LiteSVM with base, priority and inclusion fees and the fee-payer reserve, the lattice state hash, recent blockhashes, threshold transactions | `SvmExecutor::execute_slot, select_with_fees, ThresholdCache::accept_at` |
| `acrux-transport` | One QUIC endpoint per validator, each link delayed by its hub geography, Rotor relay sampling by stake, phase profiling | `LocalTransport::bind, send, link_delay, relay` |
| `acrux-node` | Algorithm 1 and Algorithm 2, genesis, the store, the gateway, rolling facts, the Fault Lab, the CLI | `Network::run, tick, begin_block, receive, try_execute, try_vote` |

`acrux-consensus` never sends, signs or reads a clock. `Votor` takes votes, certificates, local blocks and timeouts as calls and returns `Action`s (`SendVote`, `SendCertificate`, `Finalized`, `ParentReady`) for the caller to carry out; the caller verifies every vote's signature, including those inside a certificate, before it calls `receive`. Quorums count distinct identities at their stake, in integer arithmetic.

## 8.2 The runtime

`Network::new` in `protocol.rs` builds twelve validators, four in each of FRA, NYC and TYO at equal stake, and draws the epoch's 96 windows with `schedule::build_schedule`. `Network::run` gives every validator its own operating-system thread, `acrux-validator-{id}`, with a current-thread Tokio runtime. The thread owns the validator's `Node`: signing key, `SvmExecutor`, `Votor`, mempools, threshold cache, and per-slot and per-window state. Validators share no protocol state; they meet only on the wire. A multi-threaded runtime of eight workers carries QUIC sends, the hash-exchange timers, the gateway and the store's writer, and CPU-heavy steps (encoding a block, rebuilding a slice, executing a slot, signing, verifying) run on the blocking pool.

### 8.2.1 The actor loop

`NodeRunner::run` is each validator's loop. It selects over its queue of authenticated packets (one, then up to seven more before yielding), a 1 ms interval that calls `Network::tick`, and on validator 0 the traffic generator and

the gateway's commands. `Network::tick` is the clock of Algorithms 1 and 2: for each of the validator's two newest windows it starts every slot whose time has come ( `elapsed(run.start) ≥ next_slot · 50 ms` ), calls `begin_block` if the validator proposes in that window, then runs `try_execute` and `try_vote`. It also prunes: shreds 64 slots after execution, checkpoints older than two windows, and everything 64 consensus slots below finality, which the store already holds.

### 8.2.2 Ingress and egress

All endpoints deliver into one ingress task. An `Envelope` begins with its `from` as a fixed-width `u64` and the `u32` variant index of its body (Appendix D), so bytes 8 to 12 pick out a commitment without decoding the packet. A commitment is authenticated at once and recorded by `ExchangeEngine::receive_commit` on the exchange's own lock, never behind a validator's data queue. Every other packet goes to its validator's own queue, where `authenticate` checks batches of up to 128 on the blocking pool: the envelope signature, a commitment's scheduled proposer, every shred's signature and Merkle proof, both signatures of an equivocation proof, and a handover's attestations. A cache keyed by SHA-256 of the packet, and of each shred with its signer, spares a relayed shred a second check. Outbound, `Network::dispatch` signs each body once and spawns one send per recipient; telemetry goes to the `Observer`, the single consumer that projects events into the gateway's views and hands records to the store.

## 8.3 The life of a slot

This section follows one slot *s* of a window through the code, from the proposer's first byte to the validators' vote on its cblock. The protocol it implements is told in Chapter 3 and Chapter 4; Figure 8.2 places each step on the slot's clock.

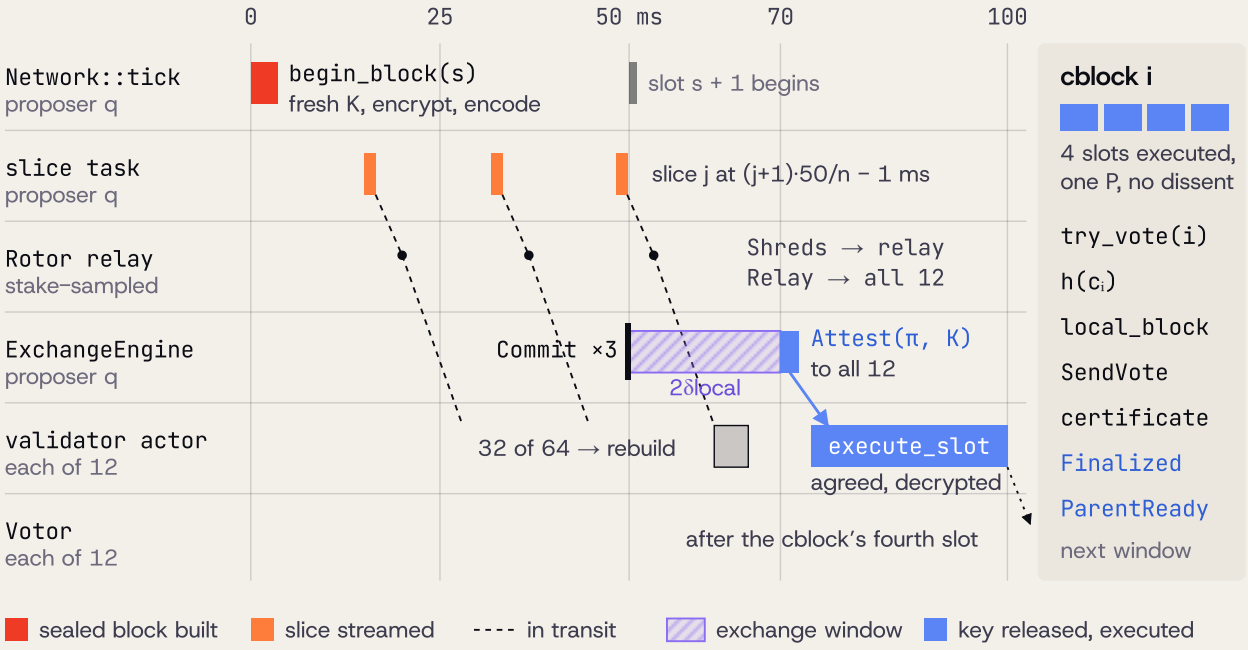



**Figure 8.2** One slot through acrux-node, drawn for a block of three slices. The axis is the proposer's slot clock; the commitment leaves at 50 ms and the exchange closes at most 20 ms after `start[s]`. Arrival and execution times at validators depend on the link (Section 8.6) and are measured in Chapter 9.

### 8.3.1 Window start

Votor's `ParentReady` action, or the genesis call for window 0, puts the window and its parent on the node's `pending_starts`; the next tick calls `start_window`, which records a `WindowRun` with the window's start instant and parent (Algorithm 2, lines 4 to 8). A proposer of the previous window also hands on to the proposer in its own role of the new window every transaction and fragment it did not get to include, as `Wire::Submit` and `Wire::SubmitFragment`. Each window starts at a `ParentReady` certified by the previous window's last cblock, so a window is never paced against a wall clock.

The first block of a window can start before `ParentReady`. `prepare_early` streams the encrypted prefix slices of the first slot once the validator has executed the previous window, with no parent in them; `complete_early` adds the final slice, which carries the parent, after `ParentReady` (Algorithm 1, lines 10 to 13).

### 8.3.2 Proposing

`begin_block(id, s)` implements Algorithm 1, lines 7 to 9. It draws a fresh 32-byte key from the operating system's generator, takes every mempool transaction and fragment whose `targetSlot` is at most $s$, and encodes them as one `BlockPayload` of at most 1 MiB. On the blocking pool it cuts the payload into chunks of 38,000 bytes, encrypts each with `encrypt_slice` and encodes it into 64 signed shreds with `encode_slice`. The final slice carries the parent reference in its `ShredMeta` when $s$ is the first slot of its window.

The commitment's hash is SHA-256 of the `BlockCommitment` (slot, proposer, H(K), parent, the slice roots in order), so one 32-byte value binds the whole ciphertext and the key.

A spawned task streams the slices across the slot: slice *j* of *n* leaves at `start + (j + 1) · 50 ms / n - 1 ms`. For each slice it groups the 64 shreds by their relay, `transport.relay(s, slice, q, index)`, and sends one `Wire::Shreds` to each relay. A relay checks that the shreds come from the scheduled proposer and broadcasts them to all twelve validators as `Wire::Relay`.

### 8.3.3 The hash exchange

`begin_block` ends by calling `ExchangeEngine::launch`, which runs Algorithm 1, lines 14 to 16 and 24 to 32, on the I/O runtime, apart from the validator's actor. Its timer wakes at the slot's start plus 50 ms. It marks the commitment sent, sets `start[s]` to the earlier of now and the first commitment already received, and sends the signed `Commit` to its co-proposers three times, 250 µs apart, for redundancy. It then waits until every co-proposer's commitment has arrived or `start[s] + 20 ms` has passed, whichever is first.

`Exchange::attestation_vector` builds $\pi$: entry $q$ is the hash of $q$'s commitment if it arrived by the deadline, and the attester's own entry is always set. The engine signs `Attestation(s, q, π, K)` and sends it to all twelve validators. On the receiving side, `receive_commit` keeps the first commitment of each proposer and its signature; a second, different commitment of the same proposer for the same slot becomes a `Wire::Equivocation` holding both signed commitments, broadcast to every validator (Lemma 2.4).

### 8.3.4 Validating and executing

A validator's `receive` places each relayed shred in the pile of its proposer and slice. A shred whose root, metadata or data differs from one already held at the same index, or whose root differs from the pile's, marks the slot as equivocated and is stored under `equivocation/block/{s}/{q}`. The 32nd shred of a pile triggers `reconstruct_verified_slice` on the blocking pool, which rebuilds the full codeword, checks that it reproduces the committed Merkle root, and returns the ciphertext. Attestations land in the slot's `attest` map.

`try_execute(id, w)` is Algorithm 2's TryExecute, run for the window's slots in order. For slot *s* it calls `agreement::agreed` on the slot's attestations (lines 53 to 59): every attestation agrees on one $\pi$, the attested set is a strict majority, every attested proposer's attestation is held, and no proposer equivocated. For every proposer $q$ in the agreed set it then checks that SHA-256 of the released key equals the key hash in the shreds, decrypts every slice with `decrypt_slice`, rebuilds the `BlockCommitment` from the shreds' roots and compares its hash with $\pi[q]$, decodes the payload, and revalidates every transaction and fragment in it. Any failure leaves the slot unexecuted with its reason recorded.

At the first slot of a window the parent is the most common parent among the agreed blocks, ties to the lowest proposer index (line 32). The executor is restored to the checkpoint of that parent, so a validator that executed a branch which was later skipped continues from the right state. Inclusion fees are checked and charged, fragments are accepted into the threshold cache (a $t$-th distinct fragment decodes its transaction), expired fragment sets are released, and `SvmExecutor::execute_slot` merges, selects and executes on the blocking pool. Every validator's state hash for the slot is compared across the net; the slot counts as checked when all twelve validators report the same hash. After the fourth slot of a consensus slot the executor and the threshold cache are checkpointed.

### 8.3.5 Voting

`try_vote(id, w)` is TryVote (lines 43 to 52). Consensus slot $i$ is voted once, in order, when its four slots are executed on one agreed set, no slot shows disagreeing vectors, an invalid block or an equivocation, no skip certificate covers it, and its timeout has not passed. At the first consensus slot of a window the validator waits for all three attestations of the first slot until $\delta_{\text{block}} + 2\delta_{\text{global}} + 2\delta_{\text{local}} = 570$ ms after the window's start (line 46). The cblock hash is SHA-256 of `(i, parent, [(s, P, π)] for its four slots, state hash)`, and `Votor::local_block` receives the block. Votor's actions become signed `Wire::Vote`s and `Wire::Certificate`s with their signed vote proofs, sent to all twelve. The proposers of a window send the last cblock's signed shreds and attestations to the next window's proposers as `Wire::Handover` (Algorithm 1, line 17).

The timeout of consensus slot $i$ in a window starting at $i_0$ is $\Delta_{\text{timeout}} + (i - i_0 + 1)\cdot\kappa\cdot\delta_{\text{block}} + \delta_{\text{block}}$ after the window's start (line 6), that is 1,000 ms for the first cblock and 1,600 ms for the fourth. On timeout `Votor::timeout` casts the skip vote.

---

## 8.4 Cryptographic choices

The paper names functions (a hash, a signature, a symmetric cipher, an erasure code, a secret sharing) and leaves their construction open. acrux-node fixes each to a construction Solana validators already run or that is standard for its job. Table 8.2 lists them; the bytes each produces are in Appendix D.

**Table 8.2** Cryptographic constructions in acrux-node and what each one carries.

| Function | Construction | Used for | Code |
|---|---|---|---|
| Hash | SHA-256 | commitments, H(K), Merkle trees, transaction ids, account hashes, blockhashes, cblock hashes | `crypto::hash` |
| Signature | Ed25519, strict verification | envelopes, shreds, commitments, attestations, votes, the threshold header | `sign_bytes`, `verify_bytes` |
| Block cipher | ChaCha20-Poly1305, a fresh 32-byte key per block | every slice of a proposer block | `encrypt_slice`, `decrypt_slice` |
| Rotor code | Reed-Solomon over GF($2^8$), 32 data and 32 coding shreds, Merkle proofs of depth 6 | slice dissemination | `encode_slice`, `verify_shred` |
| Fragment code | Systematic Reed-Solomon over GF($2^8$), $t$ of $n$ | threshold transaction ciphertexts | `threshold::reconstruct` |
| Key sharing | Shamir over GF(256), $x$ = index + 1 | threshold transaction keys | `seal_for_receivers` |
| State commitment | Lattice hash: sum of account hashes mod $2^{256}$ | the state hash every cblock carries | `SvmExecutor::refresh` |

**Ed25519.** Validator identities are Solana keypairs: the key that signs a node's commitments and votes is the identity that joins a hub on the Hub Registry (Chapter 11). Verification uses `verify_strict`, which rejects weak keys and malleable signatures, so an equivocation proof cannot be disputed by re-encoding it.

**ChaCha20-Poly1305 with H(K) in the commitment.** No key encrypts two blocks, so the nonce is derived rather than sent: SHA-256 over a domain string, the slot, the proposer and the slice index, which also keeps one block's slices apart. The associated data binds the same three numbers, so a slice cannot be moved to another slot, proposer or position. The key's hash travels in every shred and in the commitment, and a validator checks it before it decrypts: a released key opens exactly the committed block or nothing (Lemma 2.3).

**Reed-Solomon 64/32 with Merkle proofs.** Any 32 of a slice's 64 shards rebuild it. The shards are the leaves of a Merkle tree with domain-separated leaves and inner nodes, and the proposer signs each shred's metadata, index, root and data hash, so a relay forwards what it cannot forge and a receiver checks every shred alone. After rebuilding, `reconstruct_verified_slice` re-derives the root from all 64 shards, so a codeword whose coding shards disagree with its data is rejected.

**Shamir over GF(256).** A threshold transaction's key is shared byte by byte with polynomials of degree $t - 1$, share $i$ at $x = i + 1$; any $t$ shares interpolate it at $x = 0$ and fewer reveal nothing. Each receiving proposer holds one fragment and one share, and no proposer can read the transaction alone (Chapter 5).

**The lattice state hash.** `SvmExecutor::refresh` hashes each account (address, lamports, owner, executable flag, rent epoch, data) with SHA-256, and the state hash is the sum of all account hashes modulo $2^{256}$. Changing an account subtracts its old hash and adds its new one, so a slot costs one hash per touched account, never a pass over the state, and the result does not depend on the order of changes.

## 8.5 Execution on LiteSVM

`SvmExecutor` in `acrux-execution/src/execution.rs` wraps a LiteSVM instance with every Solana feature enabled, a mirror of its accounts with their hashes, the set of programs loaded at genesis, the fee schedule, the recent blockhashes and the executed transaction ids. Execution is deterministic: the same agreed slot yields the same state hash on every validator.

### 8.5.1 Admission

`validate_transaction` is the one check a transaction passes at the gateway, in a proposer's mempool, in a decrypted block and at the execution boundary: at most 1,232 bytes, no trailing bytes, a sanitized message, a recent blockhash, valid signatures, no address lookup tables, only genesis programs, and at most one compute limit and one price. Without a limit a transaction gets 200,000 CU per instruction, capped at 1,400,000. It returns the public `TxHeader` (id, fee payer, priority fee per CU, compute limit, writable accounts, target slot), and every copy must present exactly this header, so a header can never redirect fees or locks away from the signed message.

### 8.5.2 One slot

`execute_slot(s, items)` carries out Algorithm 2, lines 38 to 42, with Algorithm 4 as the merge.

1. It sets the SVM clock to slot $s$, with one epoch every 432,000 slots and the Unix time advancing one second every 20 slots.
2. It revalidates each item, drops ids already executed, and merges copies of one transaction by id, collecting the proposers that carried it.
3. `sort_transactions` orders by priority fee per CU, highest first, then by compute limit, then by id; `select_with_fees` places each transaction on the earliest free position of the four tracks of 2,000,000 CU whose writable accounts do not collide (Chapter 5). The priority fee, the compute limit times the price rounded up to whole lamports, is charged in the same pass and split evenly among the carrying proposers.
4. For each selected transaction it charges the base fee, 5,000 lamports per signature, burns half and splits the rest among the carrying proposers. LiteSVM charges Solana's native fees itself, so the executor credits them to the fee payer before simulating: the Acrux fees are charged once.

5. A transaction whose fee payer would end below the reserve $\rho$ of 1,000,000 lamports fails with its fees kept, and its account changes are discarded. Otherwise its post-accounts are written back and each one's hash is refreshed.
6. `advance_blockhash` appends the slot's blockhash, SHA-256 of the previous blockhash, the state accumulator and the slot, and returns the result with the state hash.

Inclusion fees are charged before the slot by `charge_inclusion`: 1,000 lamports for each copy or fragment a proposer included, paid to that proposer. Per window, a fee payer's inclusion fees to one proposer are capped at $\rho/(2k)$, one sixth of the reserve, which `validate_inclusions` checks for the whole slot first. With these fees a plain send to one proposer costs 6,000 lamports and a send to all three or a 2-of-3 threshold send costs 8,000 (Chapter 9).

### 8.5.3 Recent blockhashes and replay

The executor keeps one blockhash for every executed slot of the last 1,200 slots, one minute, and a transaction may reference any of them. Each executed id is kept with the slot of the blockhash it referenced, for as long as that blockhash is valid: a replay of the same signed bytes is dropped while it could still land, and the id is forgotten once it no longer could. `latest_blockhash`, which the gateway's `getLatestBlockhash` returns, is the blockhash from eight slots back, so every validator already holds it however far its execution trails the gateway's.

## 8.6 Transport

`LocalTransport::bind` in `acrux-transport/src/transport.rs` opens one QUIC endpoint per validator, validator $i$ on `--base-port` + $i$ (4950 to 4961 by default), with a self-signed certificate, up to 2,048 concurrent unidirectional streams, a 60 s idle timeout and a 10 s keep-alive. Every ordered pair of validators completes its TLS handshake before any slot clock starts, so a protocol send never pays for a handshake. Each message is one unidirectional stream of at most 1 MiB, and even a message a validator sends to itself crosses its own endpoint.

`send` holds each message for its link's delay before writing it. `link_delay` draws a deterministic delay from the run seed, the two endpoints and the message's sequence number on that link: inside a hub, 1 to 4 ms, below $\delta_{\text{local}}$; between hubs, the great-circle distance at two thirds of the speed of light in fibre (199,861.638 km/s), plus 8 ms for switching and hosts, plus up to 2 ms of jitter. Table 8.3 gives the result for the three genesis hubs.

Table 8.3 One-way link delays between the genesis hubs, from `link_delay` and the hubs' coordinates in `Network::new`.

| Link | Great circle | Fibre at 2/3 c | One-way delay |
|---|---|---|---|
| Inside a hub | | | 1.000 to 4.000 ms |
| FRA to NYC | 6,188.5 km | 30.964 ms | 38.964 to 40.964 ms |
| FRA to TYO | 9,361.1 km | 46.838 ms | 54.838 to 56.838 ms |
| NYC to TYO | 10,875.4 km | 54.415 ms | 62.415 to 64.415 ms |

`relay(slot, slice, proposer, shred)` is Rotor's relay choice: a point drawn from the seed and the four numbers, mapped onto the validators by cumulative stake. Every shred of every slice gets an independent, stake-weighted relay, so relays sit in every hub and no single validator carries a whole slice.

`Profiling` times named phases (`transport.timer_overshoot`, `verify.signature`, `exchange.commit_to_attest`, `rotor.reconstruct`, `validator.merge_select_execute` and others) on a monotonic clock without touching any deadline, keeping each phase's count, total, maximum and last 4,096 samples for `profile.json`. The process runs at high priority with raised thread priority, so a busy host does not delay a commitment past $2\delta_{\text{local}}$.

## 8.7 The store

`storage.rs` keeps a run in one redb database, `net-{start}.redb` in the `--store` directory, with one table of string keys (`slot/{s}`, `transaction/{id}`, `equivocation/block/{s}/{q}`, `equivocation/attestation/{s}/{q}`) and JSON values. Each start is a fresh genesis with a fresh store, and the previous run's store is kept beside it.

Records reach `write_batches` on a blocking worker through a channel, so writes never touch an actor; it commits up to 256 queued records in one atomic transaction and deletes every record whose last write lies more than `ACRUX_RETAIN_SLOTS` slots, 24,000 or twenty minutes of the net, behind the newest slot. The gateway reads older slots from the store. On stop the node writes `last-run.json`, the run's report, and `profile.json`.

## 8.8 The gateway

`gateway.rs` serves three axum listeners on `ACRUX_BIND` (127.0.0.1, or 0.0.0.0 in a container): JSON-RPC on `--gateway-port` (4930), the event stream on `--ws-port` (4931), and the private port `--metrics-port` (4989)

with `/metrics`, `/health` and the faucet's grant route `/internal/faucet`, which only the API calls and which never gets a public name. Every shape is `@acrux/contract`'s; the full method, event and error reference is Appendix B.

### 8.8.1 JSON-RPC

Requests are JSON-RPC 2.0 with an id and an object of params, at most 1 MiB. Reads and submissions draw on separate token buckets per client address (`limits.rs`); behind a proxy the client is the last `X-Forwarded-For` entry, the one the proxy appended. Errors carry the contract's envelope: a code, a message, whether it is retryable, and the stage.

Its methods read the net now (`getSnapshot`, `getCurrentWindow`, `getSchedule`, `getCounters`, `getFacts`), slots (`getSlot`, `getRecentSlots`), accounts and transactions (`getLatestBlockhash`, `getBalance`, `getTransaction`) the executed state and blocks a follower re-executes from (`getExecutionSnapshot`, `getSlotBlocks`), the demo pool (`getDemoPool`, `getQuote`), and send (`submitTransaction`, `submitFragment`).

`submitTransaction` validates the signed bytes against validator 0's executor, checks that the presented header is the one the bytes derive, and that the target slot is one of the next 64 slots and not yet started (`submission::validate_target`). It then sends `Wire::Submit` to the proposers of the target window in the roles the client chose. `submitFragment` verifies the fragment, its key share and the fee payer's header signature with `FragmentSubmission::verify`, then sends it to the one proposer in its receiver role. Both are idempotent: a retry with identical content returns the same answer and is not sent twice, and a different payload under the same id is refused as `IDEMPOTENCY_CONFLICT`.

### 8.8.2 The event stream

Every protocol step emits an event, from each slice, commitment arrival and attestation to the agreed set, the merge, execution, every vote and certificate and each transaction's trail. An event is `{ version, sequence, streamId, type, slot, anchor, atUs, data, node }`: `anchor` names what `atUs` counts from and `node` is the validator that observed it. A client subscribes to all events, a range of slots or one transaction, and may resume after a sequence number within the last 24,000 events; otherwise it gets a `stream.resync` with a full snapshot.

### 8.8.3 Metrics and facts

`/metrics` serves four Prometheus series: slots, finalized cblocks, skips and the average commit spread. `getFacts` serves the net's rolling measurements from `facts.rs`, kept by validator 0's actor on the node clock over 12,000 slots, about ten minutes: slot pacing, handover per hub link with that link's floor, commit spread, attestation arrival and cblock finalization, each as nearest-rank p50 and p99. `rotor_floors` derives each link's floor from the transport's own link model: one slot, plus the time until the next hub holds two hubs' votes for the 60% certificate (Chapter 9).

## 8.9 The CLI

The binary is `acrux`, built from `acrux-node/src/main.rs` with clap.

Table 8.4 The `acrux` commands. Flags and outputs are in Appendix B.

| Command | Does |
|---|---|
| `acrux net up` | Acrux Net: twelve validators in FRA, NYC and TYO, the gateway and the store, from the genesis file |
| `acrux cluster up --hubs FRA,NYC,TYO` | a local net on the hubs given |
| `acrux keygen, acrux identities` | a Solana keypair file; the validators' identity keypairs, each with its hub |
| `acrux send` | a plain or *t*-of-*n* transfer, followed to finality |
| `acrux node follow` | follows a net from its latest finalized cblock: checks every attestation's signature and Agreed(*s*), decrypts, merges and re-executes every finalized slot, and checks the state hash and each cblock's certificate |
| `acrux hub register, join, leave, move, release` | the Hub Registry's instructions, signed and paid by the validator identity |
| `acrux schedule` | Algorithm 3 for hubs given as JSON on stdin |
| `acrux lab run` | one Fault Lab scenario on a sandbox of twelve validators |

## 8.10 Genesis

`genesis.rs` loads Acrux Net's genesis from `backend/chain/genesis/acrux-net.json` (`--genesis`). The file holds a seed from which every genesis keypair derives as an Ed25519 key from SHA-256(seed ‖ "/" ‖ label), and the loader checks each derived address against the file. It carries:

- the `demo_pool` program, an Anchor constant-product pool built in `backend/chain/`, at `CUBR9LNdTwxwp5ARBQj1EopfuH8dr8u4i7DsXcb8HbHN`;
- two mints with 6 decimals, AXA (Acrux Demo A) and AXB (Acrux Demo B), and the pool's two vaults holding 1,000,000 AXA and 2,000,000 AXB, with a 30 bps fee;
- the `hub_registry` program at `ABjgHL8TcjxUjx6vfwsVQyvxiXR6pFgMkeESgwK6CbDm`, with its Registry account, the hubs FRA, NYC and TYO and the twelve validators' memberships. `Genesis::registry_hubs` reads them back, and `Network::new` derives from them each validator's hub and place, the link delays between validators, and the input to Algorithm 3 (Section 11.2);
- the faucet, with 100 ACRUX and a token account for each mint. A grant is one signed transaction: an ACRUX transfer and, per mint, the receiver's associated token account and 100 tokens.

Further programs load with `--program ADDRESS=PATH`. The validators' identities derive from `ACRUX_IDENTITY_SEED` on a public net, a secret, since those identities hold the net's Hub Registry memberships, and from the run's seed otherwise. A public host makes its genesis from its own secret seed with `backend/chain/scripts/genesis.mjs`.

## 8.11 The API and the contract

`backend/api/` is the service the site talks to: TypeScript on Node 22, Hono 4.13, and Prisma 7.10 over Postgres through the `pg` adapter. In `MODE=node` (`node-app.ts`) every route answers from Acrux Net's gateway, the Hub Registry's read model in Postgres, or the Fault Lab runner; in `MODE=fixture` it serves the contract's fixtures, one for every screen state.

Table 8.5 The API's routes by source. Request and response shapes are in Appendix B.

| Source | Routes |
|---|---|
| Hub Registry read model | `GET /v1/hubs, /v1/hubs/:code, /v1/registry/config, /v1/registry/members/:identity` |
| Algorithm 3 | `GET /v1/schedule` |
| The gateway | `GET /v1/facts, /v1/transactions/:id` |
| Faucet | `GET /v1/faucet/:address, POST /v1/faucet` |
| Fault Lab | `GET /v1/lab/scenarios, GET and POST /v1/lab/runs, GET /v1/lab/runs/:id, /events, /stream` |
| The service | `GET /health, /openapi.json, /v1/spec` |

**Registry indexer.** One `RegistryIndexer` runs per chain, Acrux's registry through Acrux's Solana-compatible RPC and Solana mainnet's, and marks each hub with its chain. Each reads, each tick (2 s on Acrux and locally, 30 s on mainnet), the epoch and the registry program's newest signature. Only when either changed does it read the program's accounts, decode them by their Anchor discriminators, convert hub ECEF coordinates to latitude and longitude, place each hub in its SIMD-0675 region, join members with their stake, compute proposer-stake caps through `acrux schedule`, and write hubs and members to Postgres in one transaction.

**Schedule preview, faucet, Fault Lab.** The preview spawns `acrux schedule` with the hubs on stdin (seed `acrux-schedule-v1`, 96 windows), so preview and net share one code path. Faucet grants are counted per key and per hashed address over a rolling window, 3 and 10 in 24 h on the public host, each 2 test ACRUX plus 100 AXA and 100 AXB, signed and queued by the node through `/internal/faucet`. `NodeLab` runs one `acrux lab run` at a time on QUIC ports 5016 to 5027 and stores its `events.jsonl` and `outcome.json` in Postgres, so every run replays, also as server-sent events that resume from `Last-Event-ID`.

`@acrux/contract` 0.1.8 holds the zod schemas and types for the API, the JSON-RPC and every event, the clients `createClient` and `createGatewayClient`, the fixtures, and an `openapi.json` that carries the JSON-RPC and the event stream as `x-acrux-json-rpc` and `x-acrux-event-stream`. Its shapes follow the protocol's objects (`Window`, `Slot`, `ProposerBlock`, `Commitment`, `Attestation`, `Cblock`, `Certificate`, `Transaction`, `TxHeader`, `Facts`, `Hub`, `LabRun`, `SpecMap`), and its conventions hold everywhere: amounts and stake as decimal strings, `priorityFeePerCu` in micro-lamports per CU, hashes in lowercase hex, keys and signatures in base58, proposer and track indices from 1, and a null entry of `pi` for the paper's ⊥.

## 8.12 Build and test

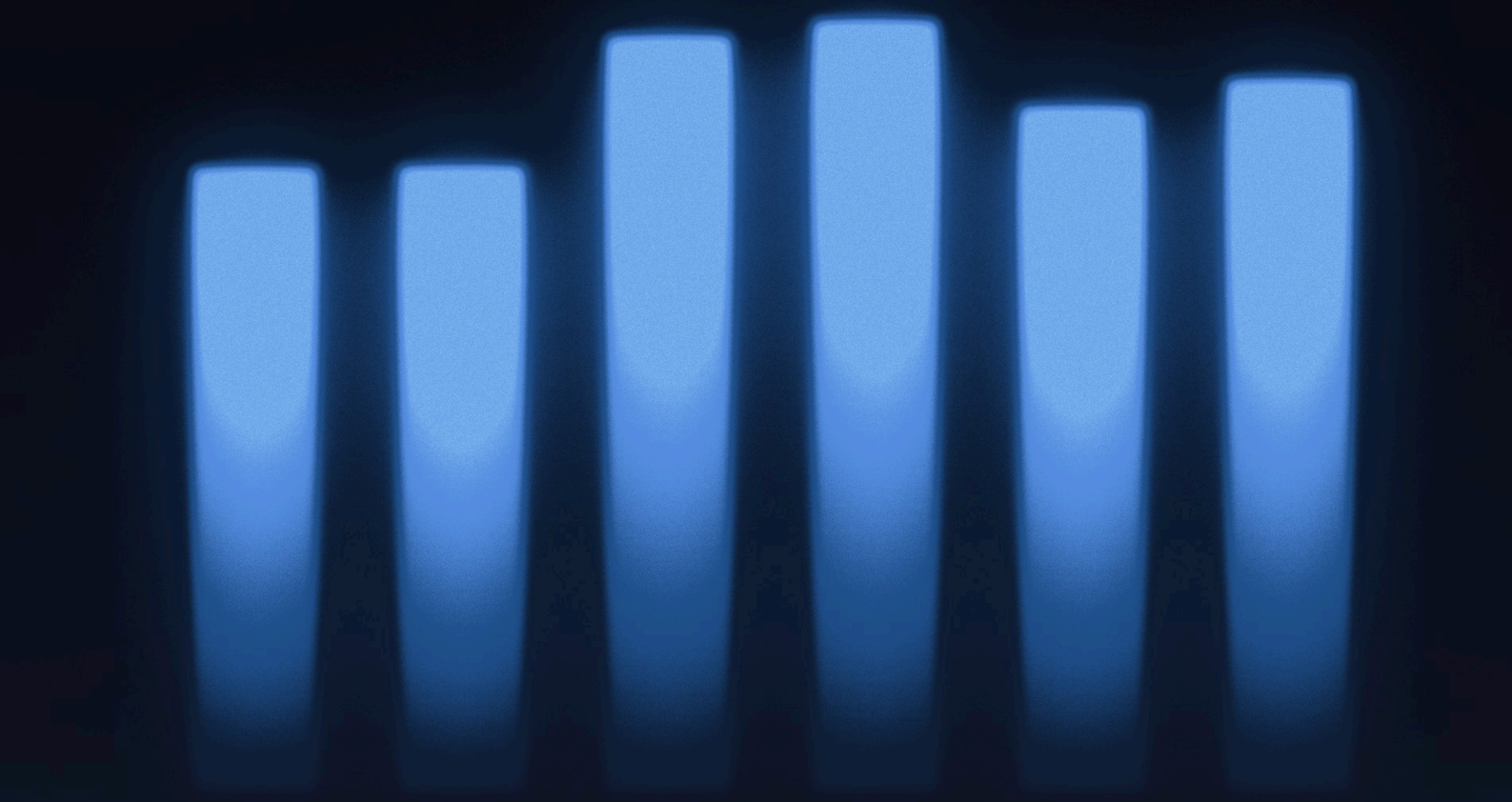
```
# Windows, from backend/node (reuses the static OpenSSL in
node/target)
powershell -ExecutionPolicy Bypass -File scripts/build.ps1 `
  --release
powershell -ExecutionPolicy Bypass -File scripts/build.ps1 `
  test --release --workspace

# Linux, as the host image's Dockerfile does
cargo build --release -p acrux-node

# Acrux Net, a 600 s run, one Fault Lab scenario
./target/release/acrux net up
./target/release/acrux net up --run-seconds 600
./target/release/acrux lab run --scenario last-look \
  --seed acrux-last-look-1 --out <dir>
```

The build is `cargo --locked` against the committed `Cargo.lock`, and the workspace carries 21 unit tests beside the code they test: eight for Votor's votes, fallbacks, certificates and parents, one rebuilding an encrypted slice from parity shreds and rejecting tampering, and the rest for exact fees and deterministic state, the reserve, checkpoints across branches, Algorithm 4's contention and gap filling (the paper's Figure 5), threshold reconstruction, Agreed(s), the exchange's inclusive 20 ms deadline, submission rules and Algorithm 3's draw against the fixture the API shares. The Fault Lab's six scenarios run the whole net against each lemma (Chapter 10), and `backend/spec/map.json` ties each line of the paper to its code (Appendix E).

600 S · 12 VALIDATORS · 50 TX/S
2,640 CBLOCKS FINALIZED · 0 SKIPS
SLOT PACING P99 51.3 MS



9

# Acrux Net, measured

Twelve validators in FRA, NYC and TYO, real geographic delays on the wire, fifty transfers a second. Six hundred seconds, every cblock finalized.

Acrux Net is the Acrux chain: twelve genesis validators in three hubs, the paper's parameters, real geographic delays on every link. This chapter describes the net, how it measures itself, and what a 600-second run and a run with a proposer down recorded.

## 9.1 The net

Acrux Net is the chain as its genesis configures `acrux-node` (Chapter 8). The same configuration starts by one command from `backend/node/`:

```
.\target\release\acrux.exe net up
```

It runs the hubs FRA, NYC and TYO with four validators each. The hubs, their places and their members are accounts of the chain's own Hub Registry, loaded at genesis; the node derives the link delays and the schedule from them (Section 11.2). Every window of $w = 16$ slots belongs to one hub, and Algorithm 3 (Chapter 6) draws $k = 3$ of that hub's four members as the window's proposers Q1, Q2 and Q3. The other nine validators of the window vote and relay. All twelve execute every slot and vote on every cblock.

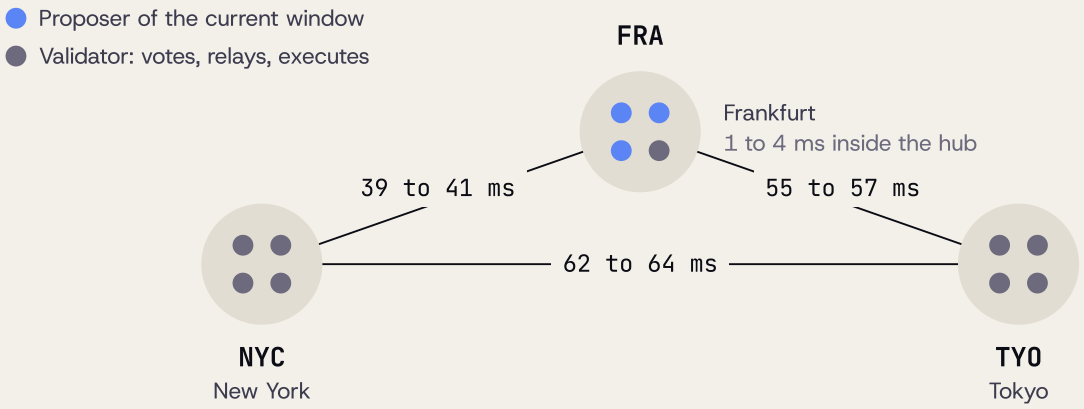



**Figure 9.1** Acrux Net. Twelve validators, four per hub; in a window of the FRA hub three of its members propose. Each link carries the one-way delay of its cities' great-circle distance at two thirds of light speed in fibre, plus 8 ms.

The delays are applied on the wire. The transport (`acrux-transport`) holds one QUIC endpoint per validator and delays every packet by the geography of its link: inside a hub 1 to 4 ms, within $\delta_{\text{local}} = 10$ ms; between hubs the cities' distance in fibre plus 8 ms. A commitment from Frankfurt to its co-proposers takes the intra-hub delay; a shred relayed to Tokyo takes the inter-hub one. Every figure in this chapter is a product of these delays and the protocol, measured on the node clock.

Table 9.1 The configuration of Acrux Net (`acrux net up` defaults, `backend/node/README.md`).

| Setting | Value |
|---|---|
| Validators | 12, four per hub in FRA, NYC and TYO |
| Proposers per window | $k$ = 3, drawn by Algorithm 3 from the window's hub |
| Link delay, one way | FRA–NYC 39 to 41 ms; FRA–TYO 55 to 57 ms; NYC–TYO 62 to 64 ms; inside a hub 1 to 4 ms |
| Traffic | 50 generated transfers a second (`--traffic-rate`), each sent to the three proposers of a slot three ahead |
| Genesis | The demo constant-product pool program, its AXA and AXB mints and vaults, the faucet; a fresh genesis at every start |
| Gateway | JSON-RPC on 4930, event stream on 4931; Prometheus metrics, health and the faucet's grant route on 4989 |
| Node-to-node | QUIC on 4950 to 4961, one port per validator |
| Store | The last 24,000 slots (`ACRUX_RETAIN_SLOTS`) |

The generated traffic is plain transfers between a ring of funded senders. Each carries a compute-unit limit of 20,000 to 24,000 and a priority price of 100 to 1,700 micro-lamports, so the merge orders real fee competition (Chapter 5). The genesis also loads the demo pool, so a browser key funded by the faucet can swap AXA for AXB in one instruction (Chapter 12).

### 9.1.1 The public host

Acrux Net goes public on one dedicated server (`DEPLOY.md`): the twelve validators, the gateway, the API with its Fault Lab sandboxes, Postgres and Caddy, from `backend/deploy/host/`. The chain's Hub Registry is part of its genesis, and the same program runs on Solana mainnet (Chapter 11). Anyone uses the net through three public names: `rpc.` for JSON-RPC, `events.` for the event stream and `api.` for the API and its faucet. Only ports 22, 80 and 443 are open; the node's QUIC, metrics and the API are reachable only inside Docker.

Table 9.2 Containers on the public host and their CPU weights.

| Container | Role | CPU weight |
|---|---|---|
| `acrux-net` | The 12 validators, the gateway on 4930 and 4931, metrics and the faucet's grant route on 4989 (never proxied) | 4,096 |
| `acrux-api` | The API on 4920, the registry read model, the faucet, Fault Lab sandboxes | 512 |
| `acrux-postgres` | Faucet grants, lab runs and their events, hubs and members | default |
| `acrux-caddy` | TLS and the three public names on 80 and 443 | default |

Public use has per-client limits: 50 JSON-RPC reads a second with bursts of 100, 5 transaction or fragment submits a second with bursts of 20, 8 event-stream connections, faucet grants of 2 test ACRUX plus 100 AXA and 100

AXB at most 3 per key and 10 per IP in 24 hours, and 6 Fault Lab runs per IP an hour with at most 10 waiting or running. The gateway and the API read the client from the last `X-Forwarded-For` entry, the one Caddy appends.

## 9.2 How the net measures itself

All twelve validators of Acrux Net run in one process, so every timestamp in a measurement is read from one monotonic clock: a commitment sent in Frankfurt and an attestation received in Tokyo are compared without clock synchronization. Once a second, validator zero folds the latest commitments, attestations and finalizations into rolling rings (`backend/node/crates/acrux-node/src/facts.rs`), and the gateway serves them as `getFacts`: the last ten minutes of the net. The API forwards the same object at `GET /v1/facts`, and the site reads it live.

| | |
|---|---|
| **Slot pacing** | The gap between the earliest commitments of two consecutive slots in the same window. The ring keeps 12,000 gaps, about ten minutes of 50 ms slots. |
| **Handover** | The same gap at a window boundary, from the last slot of one window to the first of the next, keyed by the two windows' hubs (`FRA-NYC` is an FRA window followed by an NYC window). The last 1,000 per link, each with its floor (Section 9.5). |
| **Commit spread** | At proposer Q1, the time from the first to the last commitment of a slot it received inside the $2\delta_{\text{local}}$ exchange. |
| **Attestation arrival** | When each attestation reaches each validator, in every hub, relative to the slot's first commitment. |
| **Cblock finalization** | From the first slot of a cblock to the cblock's finalization at validator zero. |

Percentiles are nearest rank: p50 is the median sample and p99 the smallest sample at or above 99% of the ring. When the net stops, it writes `last-run.json` into its store with the whole run: counts, percentiles and maxima, the state hashes and the last executed slot of every validator. The figures below come from two such reports, `backend/node/reports/acrux-net-600s.json` and `proposer-down-120s.json`, and from `getFacts` at the end of the 600-second run.

## 9.3 Six hundred seconds

The reference run is `acrux net up --run-seconds 600`: the twelve validators on one machine, fault free, at 50 transfers a second, measured on 2026-11-16 (UTC). In 600 s the traffic generator sent 30,001 transfers.

**2,640**
Cblocks finalized, every cblock of the run

**0**
Skipped cblocks

**10,564**
State hash checks, 0 mismatches

**2** slots
Execution skew between validators, at most

Every cblock the run produced was finalized, 2,640 of them, and none was skipped. All twelve validators ended on slot 10,563 with the same state hash, `85b0d390…dccd3`.

Table 9.3 The 600-second run. Rolling figures are `getFacts` at the end of the run; whole-run figures are `last-run.json`.

| Measure | Samples | p50 | p99 | Maximum |
|---|---|---|---|---|
| Slot pacing within a window, `getFacts` | 9,893 | 50.0 ms | 51.3 ms | |
| Slot pacing within a window, whole run | 9,905 | 50.0 ms | 51.7 ms | 53.4 ms |
| Commit spread | 10,566 | 4.7 ms | 12.0 ms | 15.4 ms |
| Attestation arrival after the block | | 48.5 ms | 76.1 ms | |
| Cblock finalization | 2,640 | 324 ms | 398 ms | 414.9 ms |

## 9.4 A slot, measured

A slot is 50 ms. The proposers stream their sealed blocks through Rotor for the length of the slot and send their signed commitments with the last slice, at the slot's end (Algorithm 1; Chapter 3). The exchange window of $2\delta_{\text{local}}$ = 20 ms opens at the first commitment; each proposer attests, releasing its key, once every commitment is in or the window closes. Figure 9.2 lays the measured figures over that budget.

**50.0** ms
Slot pacing p50

**51.3** ms
Slot pacing p99

**12.0** ms
Commit spread p99, inside the 20 ms window

**48.5** ms
Attestation arrival p50, every hub

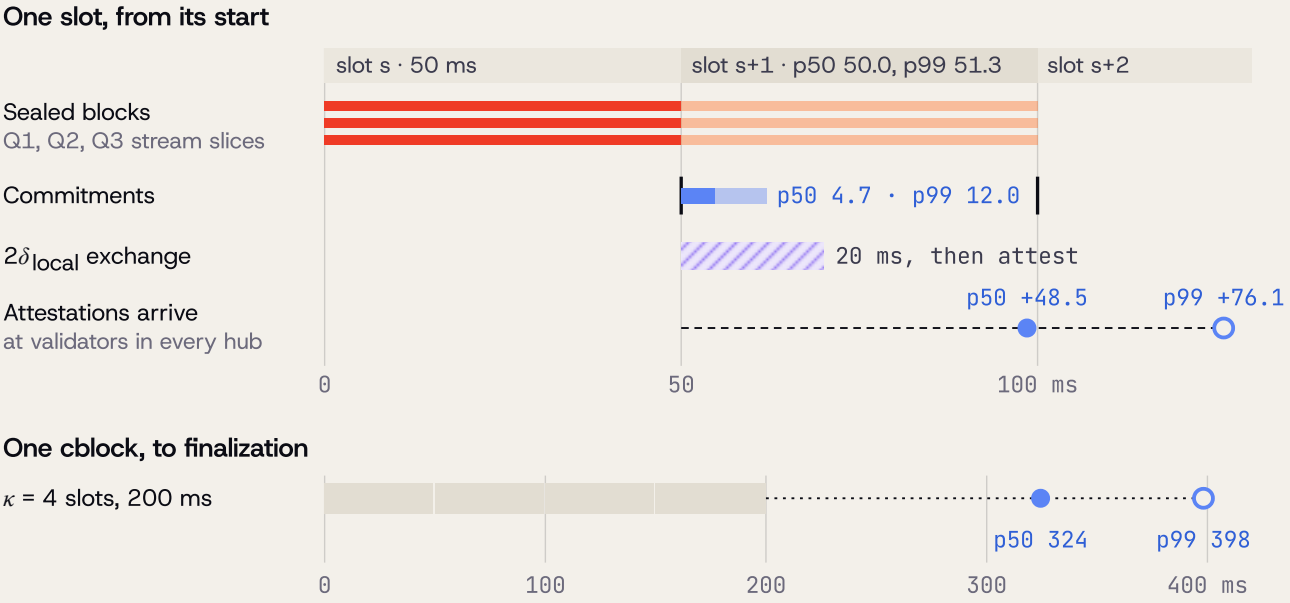



**Figure 9.2** The timing budget of a slot and of a cblock, at scale, with the 600-second run's percentiles. Commitments go out at the slot's end; the commit spread stays inside the hatched 2δlocal window; attestations are in transit (dashed) to every hub. Times on the lower axis run from the cblock's first slot.

**Pacing.** A window's slots follow one another at a median of 50.0 ms and a p99 of 51.3 ms; over the whole run, 9,905 gaps had a p99 of 51.7 ms and a maximum of 53.4 ms. The node runs at high process priority with raised thread priority, so a busy machine does not delay a commitment past $2\delta_{\text{local}}$.

**Commit spread.** Proposers of one hub finish their blocks together: from the first to the last commitment Q1 received, a median of 4.7 ms and a p99 of 12.0 ms, with a whole-run maximum of 15.4 ms. Even the maximum sits inside the 20 ms window, so in the fault-free run the commitments of a slot land inside its exchange and the slot is agreed on `P = {1, 2, 3}`.

**Attestations.** Keys leave the hub in attestations as soon as the exchange closes and reach validators in all three hubs 48.5 ms after the slot's first commitment at the median, 76.1 ms at p99. The arrivals include the validators of the farthest hub, 62 to 64 ms away on the NYC–TYO link, on top of the exchange.

**Finalization.** A cblock is four slots, 200 ms. From its first slot to its finalization took a median of 324 ms and a p99 of 398 ms: the cblock's own 200 ms, then the last slot's attestations, decryption and execution, and the votes that make its certificates. Every one of the 2,640 cblocks finalized within 414.9 ms of its start.

## 9.5 Window handovers

At a window boundary the next hub's proposers cannot begin until the previous window is behind them. Their first slot starts at ParentReady, which needs a notarization of the previous window's last cblock: 60% of the stake,

which on Acrux Net is the votes of two hubs (Chapter 4). A hub votes once it holds the slot's attestations and 32 of each slice's 64 shreds, and Rotor samples its relays by stake from every hub, so a shred reaches a hub through relays in the others. The earliest the next window's first block can complete is therefore one fixed figure per pair of hubs, computed from the link model by `rotor_floors` in `facts.rs`:

> **The floor of a handover is 50 ms plus the arrival of the second hub's votes for the 60% certificate, since Rotor's relays sit in every hub.**

The two hubs and the paths through them make the floor depend on the link. Inside one hub (FRA–FRA, NYC–NYC) the floor is 134.9 ms; TYO–TYO is 166.7 ms, since both of Tokyo's inter-hub links are long and the second voting hub is far. Figure 9.3 sets each link's measured handover against its floor; Table 9.4 lists them.

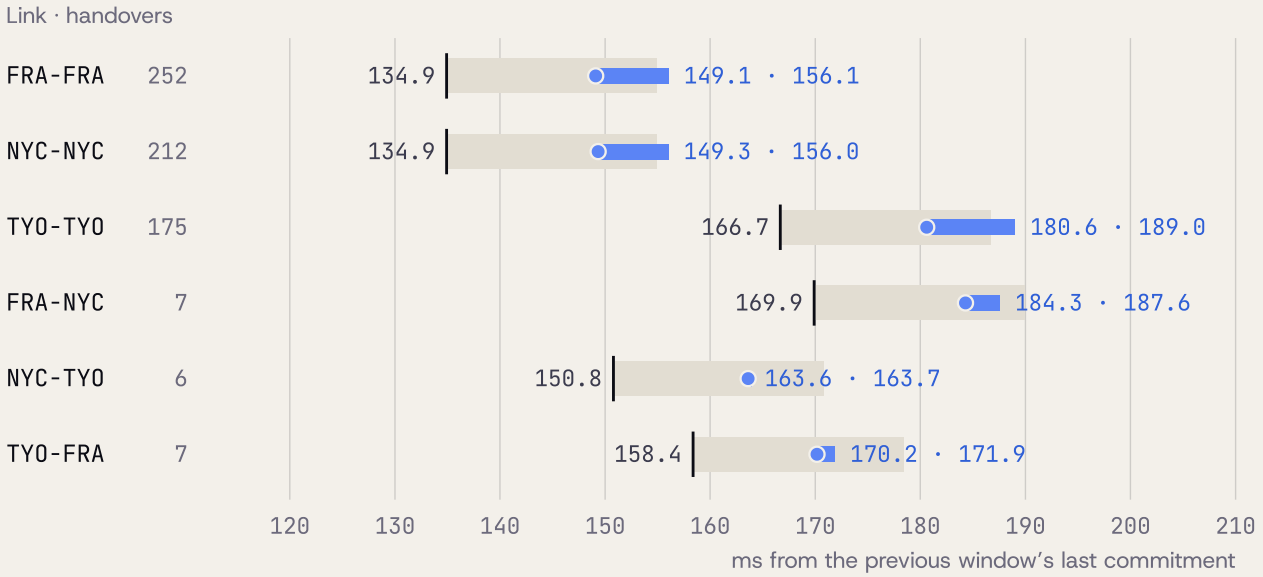



**Figure 9.3** Window handover per hub link, 600-second run. The black tick is the link's floor, the shaded band the 20 ms above it; the blue dot is p50 and the bar runs to p99. The count is the handovers of that link in the last ten minutes.

**Table 9.4** Window handovers per hub link, from `getFacts` at the end of the 600-second run.

| Link | Handovers | p50 | p99 | Floor | p99 over floor |
|---|---|---|---|---|---|
| FRA-FRA | 252 | 149.1 ms | 156.1 ms | 134.9 ms | 21.1 ms |
| NYC-NYC | 212 | 149.3 ms | 156.0 ms | 134.9 ms | 21.1 ms |
| TYO-TYO | 175 | 180.6 ms | 189.0 ms | 166.7 ms | 22.3 ms |
| FRA-NYC | 7 | 184.3 ms | 187.6 ms | 169.9 ms | 17.7 ms |
| NYC-TYO | 6 | 163.6 ms | 163.7 ms | 150.8 ms | 12.9 ms |
| TYO-FRA | 7 | 170.2 ms | 171.9 ms | 158.4 ms | 13.5 ms |

Most handovers stay inside a hub, since the hub schedule comes in runs of consecutive windows (SIMD-0675's regional runs in Algorithm 3, Chapter 6): 639 of the 659 handovers were FRA–FRA, NYC–NYC or TYO–TYO. The 20 hub changes went FRA to NYC, NYC to TYO and TYO to FRA.

### 9.5.1 The pacing bar

The project set its pacing bar before the measurement ( `PLAN.md` , Decisions): slot pacing p99 within a window at most 55 ms, and each handover link's p99 within 20 ms of its floor, measured and published per link with the floor.

- **Within a window:** p99 51.3 ms against the 55 ms bar, 3.7 ms under it; the whole run's maximum, 53.4 ms, is under it too.
- **Handovers:** FRA–NYC at 17.7 ms over its floor, TYO–FRA at 13.5 ms and NYC–TYO at 12.9 ms are within the 20 ms. The three links inside a hub run 21.1, 21.1 and 22.3 ms over theirs, the figures the site and the docs publish beside each floor.

Every one of these handovers is a window that started and finalized: the 2,640 cblocks include the first cblock of every window, and no handover produced a skip.

## 9.6 One state on twelve validators

Each validator executes every agreed slot on its own SVM and computes the lattice state hash of the result (Chapter 5); Votor certifies cblocks by that hash. While the net runs, the run report compares the state hashes the validators reach at each height. The 600-second run compared 10,564 heights across the twelve validators and found 0 mismatches.

Table 9.5 Execution across the twelve validators, 600-second run.

| Field of `last-run.json` | Value |
|---|---|
| State hash checks ( `onlineStateHashChecks` ) | 10,564 |
| Mismatching heights ( `mismatchingStateHashHeights` ) | 0 |
| Execution skew between validators, at most ( `maximumExecutionSkewSlots` ) | 2 slots |
| Execution lag behind the newest slot, at most ( `maximumExecutionLagSlots` ) | 4 slots |
| Last executed slot, each of the 12 validators | 10,563 |
| Final state hash, each of the 12 validators | `85b0d3908fb7c810…` |
| Generated transfers | 30,001 |

No validator was ever more than two slots behind another, and none more than four slots, one cblock, behind the newest slot.

## 9.7 What a send costs

The SDK's cost test ( `backend/sdk/test/cost.mjs` ) sends ten of each kind of send on the live net: a 0.001 ACRUX transfer with no priority fee, through `@acrux/sdk` , timed from submit. A plain send goes in the clear to one proposer or to all three of the target slot; a 2-of-3 threshold send splits the transaction so any two of the three proposers rebuild it (Chapter 12).

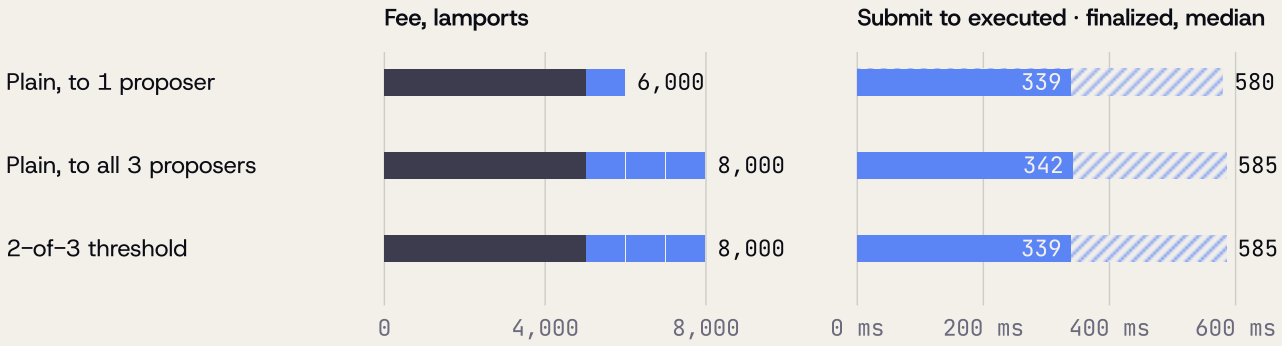



Figure 9.4 Fee and speed of three sends, ten of each, medians. The fee is the 5,000-lamport base fee (dark) plus 1,000 lamports for each included copy or fragment (blue). Solid bars run to executed, hatched to finalized.

Table 9.6 Send cost and speed on Acrux Net ( `web/lib/facts.ts` , `sendCosts` ).

| Send | Fee | Submit to executed | Submit to finalized |
|---|---|---|---|
| Plain, to 1 proposer | 6,000 lamports | 339 ms | 580 ms |
| Plain, to all 3 proposers | 8,000 lamports | 342 ms | 585 ms |
| 2-of-3 threshold | 8,000 lamports | 339 ms | 585 ms |

A 2-of-3 send costs what a plain send to the same three proposers does, 2,000 lamports more than a plain send to one, and arrives just as fast: its fragments travel in the same slot as plain transactions and are decoded the moment the second is accepted.

## 9.8 One proposer down

Theorem 2.1 says a window with a strict majority of correct proposers, the others crashed, keeps finalizing (Chapter 7). The second report runs that case on the full net:

```
acrux net up --run-seconds 120 --crash-window 5 --crash-
proposer 3
```

Proposer Q3 of window 5 sends nothing for the whole window: no slices, no commitments, no attestations. Q1 and Q2 close every exchange without its commitment and attest it as missing, so each slot of the window is agreed on `P = {1, 2}`, and the window continues on two proposers.

| 0 | 526 | 2,107 | 51.7 ms |
|---|---|---|---|
| Skipped cblocks in 120 s | Cblocks finalized | State hash checks, 0 mismatches | Slot pacing p99 within a window |

The run finalized 526 cblocks and skipped none, compared 2,107 state hash heights with 0 mismatches and kept execution skew at most 2 slots. Pacing inside windows held at p50 50.0 ms and p99 51.7 ms over 1,978 gaps, with a maximum of 52.5 ms: the same as the fault-free run. The first vote of the crashed window waits for the attestation deadline (Chapter 4), and the window's cblocks then finalize as each executes. The Fault Lab runs the same crash in a sandbox and records the agreed set as evidence (Section 10.5).

## 9.9 Six hundred seconds on the server

The same run on the chain's own host, an OVH Rise-S with the stack on its 16 threads, from the images and compose files that run the public chain (`backend/deploy/host/rehearsal.yml`, `REHEARSAL_SECONDS=600`), measured on 2026-11-16 (UTC) with eight Fault Lab runs going during it (`DEPLOY.md`).

| 2,655 | 0 | 50.9 ms | 321.9 ms |
|---|---|---|---|
| Cblocks finalized | Skipped cblocks; 0 state hash mismatches | Slot pacing within a window, p99 | Cblock finalization, p50 |

Table 9.7 Window handovers per hub link on the server, p99, against each link's floor (Section 9.5). Every link is within 20 ms of its floor.

| Link | p99 | Floor | Above the floor |
|---|---|---|---|
| FRA-FRA | 147.9 ms | 134.9 ms | 13.0 ms |
| NYC-NYC | 148.2 ms | 134.9 ms | 13.3 ms |
| TYO-TYO | 183.2 ms | 166.7 ms | 16.5 ms |
| FRA-NYC | 183.9 ms | 169.9 ms | 14.0 ms |
| NYC-TYO | 160.0 ms | 150.8 ms | 9.2 ms |
| TYO-FRA | 167.1 ms | 158.4 ms | 8.7 ms |

Cblock finalization ran at p50 321.9 ms and p99 390.3 ms, and all eight Fault Lab runs ended in their scenario's evidence while the chain ran beside them. On its own host the chain meets the whole pacing bar: slot pacing

within a window at p99 50.9 ms, under 55 ms, and every handover link within 20 ms of its floor.



10

# The Fault Lab

Six attacks New Constellation is built against, each run on a sandbox of twelve validators with a fixed seed, each ending in the outcome the paper proves.

The Fault Lab runs the attacks New Constellation is built against on a sandbox of twelve validators, puts one fault into one window, and ends each run in the evidence the paper's lemma predicts, read from the cluster's own records.

## 10.1 The runner

A lab run is one command of the node's binary. The API starts it for each run requested on `/lab`; anyone can start the same run locally:

```
acrux lab run --scenario last-look --seed acrux-last-look-1 --
base-port 5016 --out <dir>
```

The runner (`backend/node/crates/acrux-node/src/lab.rs`) starts a fresh cluster exactly as `acrux net up` does (Chapter 9): twelve validators in FRA, NYC and TYO, four per hub, the same genesis, each link delayed by its geography. The sandbox takes its own twelve QUIC ports from `--base-port` (5016 to 5027 on the public host) and runs for 12 s, fifteen windows of 800 ms.

One thing differs from Acrux Net: in window 4, slots 64 to 79, after the net has settled, one proposer role behaves as the scenario says. The attacker is the window's proposer Q3; in Private front-run it is receiver 1, and in Censorship Q3 is the honest proposer and Q1 and Q2 are the censors. Everything else in the run, the faulty node's other duties included, is the protocol as Acrux Net runs it.

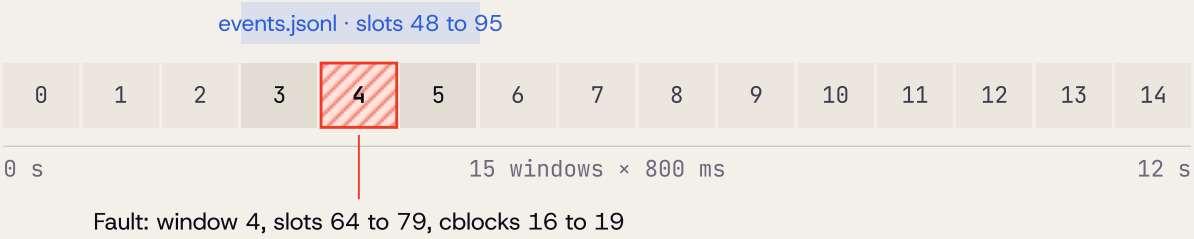



Figure 10.1 One lab run. Fifteen windows of 16 slots on a fresh sandbox; the scenario's fault is in window 4 (outlined), and every protocol event of windows 3 to 5 is written as it happens.

**Seeds.** The seed is a string. The runner hashes it with SHA-256 and takes the first eight bytes as the epoch seed of Algorithm 3, so a seed fixes which hub owns each window and which members are its Q1, Q2 and Q3; the same seed replays the same windows and proposers. Each scenario ships with its own seed (Table 10.1), and a run may name another.

**Table 10.1** The six scenarios: their seeds and parameters (`@acrux/contract` fixtures), the evidence each must end in, and what it shows.

| Scenario | Seed | Parameters | Evidence | Shows |
|---|---|---|---|---|
| `last-look` | `acrux-last-look-1` | attacker Q3, commitment delay 28,000 µs | `late_block_dropped`, `skip_certificate` | Lemma 2.3 |
| `censorship` | `acrux-censorship-1` | censors Q1 and Q2, honest Q3 | `transaction_included` | Lemma 2.2 |
| `crash` | `acrux-crash-1` | Q3 crashed at the window's start | `two_proposers_continue` | Theorem 2.1 |
| `equivocation` | `acrux-equivocation-1` | attacker Q3, 2 conflicting commitments | `slashing_proof`, `skip_certificate` | Lemma 2.4 |
| `split-commitments` | `acrux-split-commitments-1` | attacker Q3, receiver Q1 | `skip_certificate` | Lemma 2.2 |
| `private-front-run` | `acrux-private-front-run-1` | attacker receiver 1, $n = 3$, $t = 2$ | `unreadable_fragment` | Section 3.2, Lemma 2.3 |

**Records.** While the sandbox runs, the runner subscribes to the node's event stream and appends to `events.jsonl` every protocol event whose slot lies in windows 3 to 5, slots 48 to 95, in the event stream's envelope: slices, commitments and their arrivals, attestations, the agreed set or the skip, the merge, execution, votes and certificates. When the 12 s are over, the node reads the outcome from its own records, the exchange's omitted commitments, the skip certificates of cblocks 16 to 19, the agreed set of slot 64, the store's transaction records, and writes `outcome.json`:

**Table 10.2** The fields of `outcome.json` and of each piece of evidence.

| Field | Meaning |
|---|---|
| `status` | `done` when every kind of evidence the scenario expects was recorded; `failed` otherwise |
| `outcome` | The evidence's sentences, joined |
| `failure` | For a failed run, the expected evidence that is missing |
| `evidence[].kind` | `late_block_dropped`, `transaction_included`, `two_proposers_continue`, `slashing_proof`, `skip_certificate` or `unreadable_fragment` |
| `evidence[].slot, .proposer` | The slot it concerns and the proposer role it names |
| `evidence[].txId` | The marked transaction, for Censorship and Private front-run |
| `evidence[].hashes, .signatures` | The commitment hashes and Ed25519 signatures it rests on |
| `evidence[].observedFragments, .threshold` | Fragments the attacker held, and the $t$ it needed |
| `evidence[].description` | One sentence saying what it shows |

Every piece of evidence is also emitted as a `lab.evidence` event, so a viewer streaming the run sees it arrive. The sandbox's store is deleted when the run ends; the record is the two files.

**The API keeps runs.** `POST /v1/lab/runs` with a scenario (and optionally a seed) answers `201` with the run queued. The API runs one sandbox at a time (`backend/api/src/node-lab.ts`): it spawns `acrux lab run`, tails `events.jsonl` every 250 ms into Postgres under the run's own sequence, pushes each event to the run's live stream, and stores the outcome when the runner exits. Postgres then holds the whole run, so `/lab/[run]` replays it, slot by slot, long after the sandbox is gone, and a run link can be shared. The public host allows 6 runs per IP an hour and at most 10 waiting or running.

## 10.2 Reading a run by its attestations

Five of the six outcomes are decided by one rule of Algorithm 2 (Chapter 4). Each proposer's attestation for a slot carries its vector $\pi$: for each of the $k$ proposers, the commitment hash it received inside the $2\delta_{\text{local}}$ exchange, or $\perp$. A validator considers the slot agreed only when every attestation it holds carries the same vector, more than $k/2$ proposers are accepted in it, the attestation of each accepted proposer is among them, and it has seen no equivocation (`agreement.rs`, `agreed`). The agreed set $P$ is the accepted proposers. When no set is agreed, validators vote skip and the cblock is skip-certified. Figure 10.2 shows the vectors of the faulty slot in each run.

| Run | $\pi$ of Q1 | $\pi$ of Q2 | $\pi$ of Q3 | Outcome |
|---|---|---|---|---|
| Fault free | ■ ■ ■ | ■ ■ ■ | ■ ■ ■ | `P = {1, 2, 3}` |
| Last look<br>slot 64 | ■ ■ ⊥ | ■ ■ ⊥ | ■ ■ ■ | `skip` vectors differ |
| Censorship<br>slot 66 | ■ ■ ■ | ■ ■ ■ | ■ ■ ■ | `P = {1, 2, 3}` tx in B3 |
| Crash<br>slot 64 | ■ ■ ⊥ | ■ ■ ⊥ | none sent | `P = {1, 2}` |
| Equivocation<br>slot 64 | ■ ■ ■ ■ | ■ ■ ■ ■ | ■ ■ ■ | `skip` slashing proof |
| Split commitments<br>slot 64 | ■ ■ ■ | ■ ■ ⊥ | ■ ■ ■ | `skip` vectors differ |
| Private front-run<br>slot 66 | ■ ■ ■ | ■ ■ ■ | ■ ■ ■ | `P = {1, 2, 3}` tx rebuilt |

**Figure 10.2** The attestation vectors of the faulty slot. Each vector has one cell per proposer: blue, a commitment received in time; ⊥, none; ember and red, two different commitments signed by Q3. Equal vectors from a majority give the agreed set $P$; any difference, or a slashing proof, gives a skip.

The figure is the lab in one view. A fault that a proposer commits alone, by delay, by selective delivery or by double signing, can make its co-proposers' vectors differ from its own; then no set is agreed and the cblock is skipped. It cannot make a validator execute a block that a correct proposer's vector leaves out, or leave out a block a correct proposer's vector includes. A crash removes one attestation and leaves the others equal, so the slot is agreed on the two remaining proposers.

## 10.3 Last look

**The attack.** A proposer that waits to see what its co-proposers put in their blocks before fixing its own has the last look a single block producer has today. New Constellation's answer is to make waiting worthless: blocks are sealed until every commitment of the slot is fixed.

**What the faulty node does.** In window 4, Q3 sends its commitment 28 ms after it would have. A proposer's exchange task wakes at the end of its slot, 50 ms after the slot's start, and sends its signed commitment to its co-proposers; for the attacker the runner adds 28 ms to that wake-up (`exchange.rs`, `launch`). Q3 builds, seals and streams its block as usual and attests as usual.

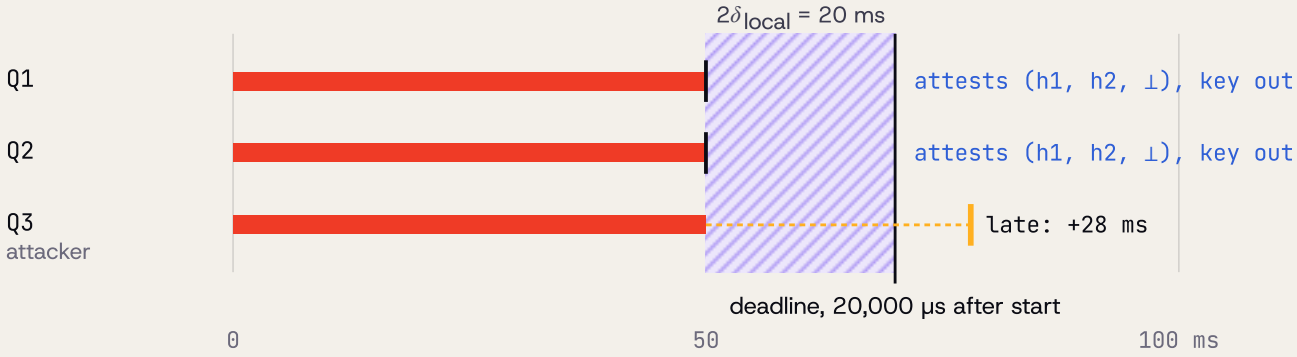



**Figure 10.3** Slot 64 of the last-look run, at scale. Q1 and Q2 commit at the slot's end and close the exchange 20 ms later; Q3's commitment (amber) comes 28 ms after the slot's end, 8 ms past the deadline, when the honest blocks are sealed and their keys already sent.

**What the protocol does.** In slot 64 Q1 and Q2 commit at the slot's end and open the exchange. At 20,000 µs after its start Q3's commitment has not arrived, so both attest it as ⊥ and release their keys in attestations with the vector (h1, h2, ⊥). Their blocks were sealed and committed before Q3 committed to anything, so whatever Q3 sees by waiting cannot change what they contain; and Q3's block, committed after their exchange closed, can only be dropped. Q3's own attestation names all three commitments, (h1, h2, h3). The vectors differ, no set is agreed, validators vote skip, and the cblock of slots 64 to 67 is skip-certified.

**The evidence.** The run ended `done` with both expected kinds:

```
{ "kind": "late_block_dropped", "slot": 64, "proposer": 3,
  "description": "Proposer 3 committed 28 ms late; its co-proposers
attested it as late (deadline 20000 us after start), so its block can
only be dropped." }
{ "kind": "skip_certificate", "slot": 64, "proposer": null,
  "description": "The late proposer's own attestation disagrees with
its co-proposers' vectors, so no set is agreed and the cblock is
skip-certified." }
```


`late_block_dropped` is read from the exchange's record of omitted commitments: a slot of window 4 in which the attacker is among the omitted proposers. `skip_certificate` is the first skip-certified cblock among 16 to 19, reported at its first slot.

**What it shows.** Lemma 2.3, proposal hiding: a malicious proposer cannot craft its block from the payload of a correct proposer's block and have both in the same cblock (Chapter 7). The run shows both halves of the proof. Until the keys are released, Q3 holds only ciphertext; once they are released, the correct attestations already fix the vector without Q3's block, so the only cblock the attack can produce is a skipped one.

## 10.4 Censorship

**The attack.** Proposers collude to keep a transaction out. With one block producer, one refusal is enough.

**What the faulty nodes do.** About six slots before window 4, the runner signs a transfer of 1,000,000 lamports with a compute-unit limit of 20,000 and a priority price of 5,000 micro-lamports, targets slot 66, the window's third slot, and sends it in the clear to all three proposers of the window (`protocol.rs`, `lab_inject`). When they build slot 66, Q1 and Q2 leave the marked transaction out of their blocks; Q3 includes it as any correct proposer does.

**What the protocol does.** Q3's block holds the transaction, and Q1 and Q2 cannot keep that block out. Q3 attests its own block, and its attestation for the window's first slot reaches every correct validator by the attestation deadline; no validator votes for any cblock of the window before it holds that attestation or the deadline has passed. All three vectors are (h1, h2, h3), slot 66 is agreed on `P = {1, 2, 3}`, Q3's block is decrypted with the other two, and the transaction is merged into the slot's order and executed in slot 66. The censors' only other move would be to make the slot skip; they cannot make it execute without the honest block.

**The evidence.** `transaction_included` is recorded when the store's record of the marked transaction says executed or finalized and its only carrier is Q3:

```
{ "kind": "transaction_included", "slot": 66, "proposer": 3,
  "txId":
  "5138be5f21c1a7116479d811996001ef48c28f3d7d0449024400c88a793878b2",
  "description": "Proposers 1 and 2 left the transaction out;
proposer 3 carried it, its block was in the agreed set, and the
transaction executed in slot 66 (finalized)." }
```


**What it shows.** Lemma 2.2, selective-censorship resistance: in a window that starts after GST, for every slot, either the blocks of all correct proposers are part of the slot or its consensus slot is skipped (Chapter 7). Here the slot executed, so the honest block, and the transaction in it, was part of it.

---

## 10.5 Crash

**The attack.** One of the window's proposers is offline from the window's start. It is the case Theorem 2.1's liveness clause is about.

**What the faulty node does.** The runner sets the node's crash window to 4 and its crash proposer to 3, the same switch `acrux net up --crash-window --crash-proposer` uses. For the 16 slots of window 4, Q3 sends nothing: no slices, no commitments, no attestations.

**What the protocol does.** In every slot of the window Q1 and Q2 close the exchange at the 20 ms deadline without Q3's commitment and attest it as $\perp$. Their two vectors are equal, (h1, h2, $\perp$), two of three is more than $k/2$, and the attestation of each accepted proposer is present, so each slot is agreed on `P = {1, 2}`. Validators cannot tell an offline proposer from a slow one, so each holds its vote for the window's first cblock until the attestation deadline, $\delta_{\text{block}} + 2\delta_{\text{global}} + 2\delta_{\text{local}} = 570$ ms after ParentReady; then it votes for that cblock and for the rest of the window as each cblock executes.

**The evidence.** The runner reads the agreed set of slot 64 at validator zero and records `two_proposers_continue` when it is {1, 2}:


```
{ "kind": "two_proposers_continue", "slot": 64, "proposer": 3,
  "description": "Proposer 3 sent nothing for the whole window; both
co-proposers attested it as missing, so slot 64 agreed on P = {1,2}
and the window continued on two proposers." }
```


**What it shows.** Theorem 2.1, NewCon correctness: after GST a cblock is finalized in every consensus slot of every window with more than $k/2$ correct proposers while its other proposers are crashed (Chapter 7). Acrux Net runs the same case at full length: 120 s with a proposer down, 0 skips (Section 9.8).

## 10.6 Equivocation

**The attack.** A proposer shows different blocks to different co-proposers, to split the validators between two versions of the slot.

**What the faulty node does.** On slot 64, the first of window 4, Q3 sends its co-proposers a second commitment for the slot, signed with the same key: the runner copies its commitment, flips every bit of the hash's first byte and signs the result (`exchange.rs`, `launch`). The two hashes in the evidence show it, `2a34e6e9…` and `d534e6e9…`, equal but for the first byte, 0x2a against 0xd5.

**What the protocol does.** A node that receives two different commitments for one slot from one proposer, both under its signature, stores them as an equivocation record and marks the slot. Two signed messages for one slot are a proof anyone can check against the proposer's identity, and a correct proposer never signs two. A validator that has seen the proof cannot consider the slot agreed, whatever the vectors say (condition (d) of Agreed; `agreed` returns no set when the equivocation flag is set), so it votes skip, and the cblock is skip-certified.

**The evidence.** `slashing_proof` carries both commitment hashes and both signatures (Table 10.3); `skip_certificate` follows for the cblock:

```
{ "kind": "slashing_proof", "slot": 64, "proposer": 3,
  "hashes": [ two, below ], "signatures": [ two, below ],
  "description": "Two commitments for one slot, both signed by the
same proposer: a slashing proof. Agreement fails and the cblock is
skipped." }
{ "kind": "skip_certificate", "slot": 64, "proposer": null,
  "description": "Validators that saw the slashing proof cannot agree
on the slot (Agreed (d)), and the cblock is skip-certified." }
```

Table 10.3 The slashing proof of the equivocation run: two commitments for slot 64, both signed by Q3's key (Ed25519, base58).

| | Commitment hash and signature |
|---|---|
| Hash 1 | 2a34e6e9373b4998b6009951cf45bbd519425463b0e3f9bd5d278094c80a9a02 |
| Hash 2 | d534e6e9373b4998b6009951cf45bbd519425463b0e3f9bd5d278094c80a9a02 |
| Signature 1 | 3Sdviey443aJBnCoC2dgNLXwxvTETHstdN1YxNXaKCaMcSHJigZQ8jxXtcvt6nowqTpx4YDJ2KThgWzNgtUL5WXv |
| Signature 2 | 2YkDQhvGBf2ha72NLBtXjyJNcZaDtMPnLeGgUzcqTAHiv3BMCRbBHkomS4bGuehYZF6a7QGz8zzt2B7D4mEiD85M |

**What it shows.** Lemma 2.4, accountable equivocation: if two correct nodes vote for different cblocks, there is a proof of equivocation blaming at least one proposer, and a correct proposer is never falsely blamed (Chapter 7). The run goes one step further than the lemma needs: the equivocation never reaches a vote, because the proof stops agreement first, and it names Q3 and no one else, with the two signed messages a slashing rule needs.

## 10.7 Split commitments

**The attack.** A proposer tries to have its block counted by some validators and not by others, by telling one co-proposer and keeping it from the other.

**What the faulty node does.** In window 4, Q3 sends its commitments to Q1 only. The runner filters its recipient list down to the co-proposer in the first role; Q3 streams its block, attests and releases its key as usual.

**What the protocol does.** In slot 64 Q1 receives Q3's commitment inside the exchange and attests (h1, h2, h3). Q2 never receives it and attests (h1, h2, ⊥). Q3 attests (h1, h2, h3). A validator holding Q1's or Q3's attestation and Q2's sees two different vectors, so no validator considers the slot agreed. A validator holding only the two equal ones does not vote either: Q2 is accepted in their vector, and the rule needs the attestation of every accepted proposer. Validators vote skip and the cblock is skip-certified.

**The evidence.**

```
{ "kind": "skip_certificate", "slot": 64, "proposer": null,
  "description": "Co-proposer 1 attested the attacker's commitment
and co-proposer 2 never received it: the vectors differ, no set is
agreed, and the cblock is skip-certified." }
```

**What it shows.** Lemma 2.2 from the other side. A byzantine proposer can make a slot skip; it cannot make validators execute different sets of blocks, and it cannot get the slot executed without a correct proposer's block. The protocol's guarantees are paid for with skips, never with a wrong block.

## 10.8 Private front-run

**The attack.** A proposer that receives a transaction reads it before its own block is sealed and puts its own trade in front.

**What the faulty node does.** About six slots before window 4, the runner builds the same transfer as in Censorship for slot 66 and seals it 2-of-3 ( `threshold::seal_for_receivers` ): it encrypts the transaction under a fresh key, Reed-Solomon codes the ciphertext into three fragments any two of which rebuild it, and splits the key into three Shamir shares with threshold 2, one fragment and one share for each of receivers 1, 2 and 3 (Chapter 5). Receiver 1 is the attacker: before anything else, it feeds the fragment it holds into a fragment cache of its own and tries to decode it.




**Figure 10.4** The private front-run. Each receiver gets one sealed fragment (ciphertext shard and key share, red: unreadable alone). Receiver 1 holds 1 of the 2 it would need; the transaction becomes readable only from the agreed blocks of slot 66, after every block of the slot is fixed.

**What the protocol does.** A fragment holds one Reed–Solomon shard of the ciphertext and one Shamir share of the key. With $t = 2$, one fragment rebuilds neither: the cache answers `Pending` with 1 accepted of a threshold of 2, and receiver 1 learns nothing of the transfer while it builds its block. The three receivers include their fragments without reading them and seal their blocks; the slot is agreed on `P = {1, 2, 3}`. Validators decrypt the agreed blocks, find the fragments, rebuild the transaction once the second is accepted and execute it in its slot. By then every block of slot 66 is committed, so no proposer can place a trade in front of it; parties who see it now can only act in later slots, ordered after it.

**The evidence.** The runner records `unreadable_fragment` from the attacker's own decoding attempt, and keeps it only if the transaction then executed:


```
{ "kind": "unreadable_fragment", "slot": 66, "proposer": 1,
  "txId":
  "9f6766c30a87fcfae0c2dc08acd85b8c2beb60e1e5b72f7fec35e018f8aef5b5",
  "observedFragments": 1, "threshold": 2,
  "description": "Receiver 1 holds 1 of the 2 fragments it would
need: neither the ciphertext nor the key can be rebuilt, so it cannot
read the transaction before the slot commits." }
```


**What it shows.** Section 3.2 of the paper, threshold transactions: a $t$-of-$n$ transaction stays hidden from any $t - 1$ colluding receivers until $t$ fragments are revealed [1]. Together with Lemma 2.3, which keeps the blocks themselves sealed until the slot's commitments are fixed, the receiver that holds the transaction first is the one party that cannot use it.

## 10.9 Six runs, six proofs

Table 10.4 The recorded runs in `backend/node/reports/fault-lab/`, all with status `done`.

| Scenario | Slot | Evidence recorded | Outcome | Lemma |
|---|---|---|---|---|
| Last look | 64 | `late_block_dropped, skip_certificate` | Q3's block dropped, cblock skipped | 2.3 |
| Censorship | 66 | `transaction_included` | Carried by Q3 alone, executed, finalized | 2.2 |
| Crash | 64 | `two_proposers_continue` | $P$ = {1, 2}, the window continues | Theorem 2.1 |
| Equivocation | 64 | `slashing_proof, skip_certificate` | Two hashes, two signatures; cblock skipped | 2.4 |
| Split commitments | 64 | `skip_certificate` | Vectors differ, cblock skipped | 2.2 |
| Private front-run | 66 | `unreadable_fragment` | 1 of 2 fragments; executed after commitment | §3.2, 2.3 |

Each scenario's answer is an outcome the paper proves, and each run ends in that outcome or in `failed` with the missing evidence named. The lab's evidence comes from the same records a validator keeps on Acrux Net: the exchange, the attestations, the certificates and the store. `backend/spec/map.json` maps each lemma to its code and its scenario (Appendix E).

hub_registry · 6 INSTRUCTIONS · 2 CHAINS
CODES OF 3 TO 8 CAPITALS · ECEF LOCATION




11

# The Hub Registry

An Anchor program where hubs are registered by code and location and joined by validator identity. On Acrux it drives the proposer schedule; on Solana mainnet, Solana's validators gather in it.

`hub_registry` is the Anchor program where hubs are registered by code and location and joined by validator identity. It runs on two chains: on Acrux, where its accounts are the input Acrux's proposer schedule reads, and on Solana mainnet, where Solana validators register and join hubs with the identity they already hold.

## 11.1 Why hubs live on chain

New Constellation needs the $k$ proposers of a window to sit within $\delta_{\text{local}}$ of each other, so that they can exchange commitments within $2\delta_{\text{local}}$ (Chapter 6). The paper does not plan where that happens. A hub forms wherever enough validators gather and disappears when they leave; the protocol only sets the conditions a hub must meet to take part in the schedule [1]. What it does fix is how hubs are known: registering a hub, joining it, leaving it and moving between hubs are on-chain transactions, and every change takes effect at an epoch boundary announced in advance.

Three properties follow from that choice, and the program provides each of them.

- **One input for every validator.** Algorithm 3 runs on every validator, seeded from the epoch, over hub memberships taken from the same snapshot as the stake. The memberships therefore live where every validator reads the same state at the same epoch: in accounts of one program.
- **Schedules fixed ahead of time.** A join, leave or move changes nothing in the current epoch. It is booked for the next boundary, and a cooldown keeps a validator from hopping between hubs to chase windows.
- **Names that stay meaningful.** A hub is registered under a short, human-readable code such as FRA, with its location as an integer ECEF point, the format of validator location registration (SIMD-0674) [14]. A deposit stands behind every code, and a hub that stays inactive past a period can be released by anyone, so codes do not stay parked.

The program is the same on both chains, byte for byte, under the same program id; Section 11.2 tells what each copy does.

**Table 11.1** The program.

| | |
|---|---|
| Program | `hub_registry` |
| Program id | `ABjgHL8TcjxUjx6vfwsVQyvxiXR6pFgMkeESgwK6CbDm` |
| Framework | Anchor 0.31.1, built for SBPF v0; the IDL and its TypeScript types beside it |
| Source | `backend/chain/programs/hub_registry/src/lib.rs` |
| Chains | Acrux, loaded at genesis, and Solana mainnet |
| Epoch | The Clock sysvar's epoch of the chain it runs on; rent from the Rent sysvar |

## 11.2 One program on two chains

**On Acrux, the registry is the schedule's source.** Acrux's genesis loads `hub_registry` with its Registry account, the hubs FRA, NYC and TYO, and the memberships of the twelve genesis validators, four per hub (`backend/chain/scripts/genesis.mjs` writes the accounts). When a node starts, it reads those accounts out of genesis (`Genesis::registry_hubs` in `backend/node/crates/acrux-node/src/genesis.rs`): each Hub by its discriminator, its code and its ECEF location turned into latitude and longitude, and each Membership in effect with nothing pending. `Network::new` (`protocol.rs`) then derives two things from them. Every validator's hub and place give the delays on every link between validators, from the distances between the hubs (Chapter 9); and the hubs with their members and regions are the input to Algorithm 3, which draws each window's hub and proposers (Chapter 6). A validator that is in no hub of the registry has no place in the net. The chain's hubs are therefore not a configuration file: they are accounts on the chain, changed by the chain's own transactions.

`acrux hub` sends its instructions to Acrux through the Solana-compatible JSON-RPC on Acrux's own endpoint; `register_hub` and `join_hub` confirm there as on any Solana cluster, paid in ACRUX.

**On Solana mainnet, the registry is where Solana's validators gather.** The same program, unchanged, lets a Solana validator register a hub or join one with the identity that already holds its stake on Solana, under the same rules, at Solana's epoch boundaries. It records where Solana's stake sits and which hubs it forms, by the paper's own rules for hubs ([1, 14]). The program is live on Solana mainnet-beta, upgradeable, deployed at slot 470,572,647, with its Registry at `4eNA25M1QpgtumzFxVkfh8nDvUFVCzTW5jtbAd1KQvcF` initialized with a 1 SOL deposit, a 500 bps registration fee, a cooldown of 1 epoch, 4 epochs of inactivity and $n_{\min} = 4$; every account and both transactions are in Table C.1.

**The API reads both.** One indexer runs per chain (`backend/api/src/registry.ts`): Acrux's registry through Acrux's Solana-compatible RPC every two seconds, with stake from Acrux's own vote accounts, and Solana mainnet's within the shared RPC budget, with stake from Solana's. Each hub is stored and served with the chain it is on, so `/hubs` shows the hubs of both chains side by side and marks each one's chain.

## 11.3 Accounts

The program keeps four kinds of account, all program-derived addresses. Their seeds make every address computable from what a user already knows: a hub from its code, a membership from the validator identity.

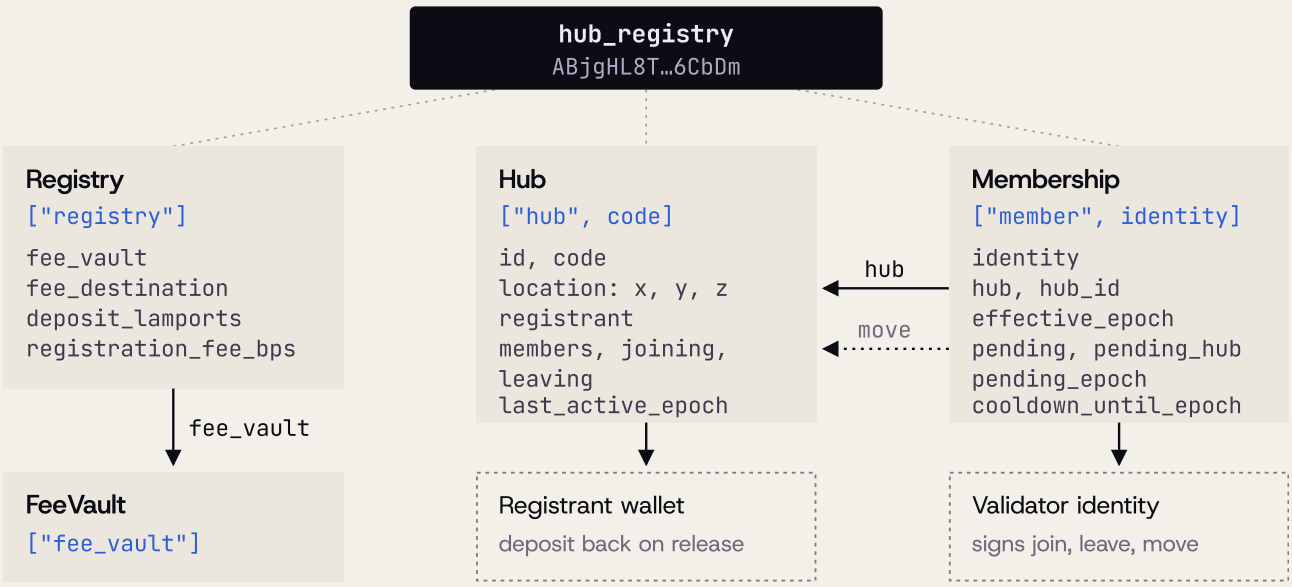



**Figure 11.1** The account graph. Each box is a program-derived address with its seeds (blue); arrows are the public keys an account stores. A membership points at its hub by address and id, and at the hub of a pending move; the wallets (dashed) are outside the program.

| | |
|---|---|
| **Registry** | One account, seed `["registry"]`. The configuration set at `initialize`: the deposit, the registration fee in basis points of the deposit, the cooldown and inactivity periods in epochs, the member floor $n_{min}$, the code length range, the fee vault and the fee destination; and `hubs_registered`, the count of hubs ever registered. |
| **FeeVault** | One account, seed `["fee_vault"]`. Registration fees accumulate in its lamports above its rent. No key signs for it; only `sweep_fees` moves them, to the registry's fixed destination. |
| **Hub** | One account per registered code, seed `["hub", code]`. Its id, its code padded to 8 bytes, its location as an ECEF point in whole metres (`i32` each), its registrant, the deposit it holds above rent, and the counts that say who is a member in which epoch: `members` in effect during `counted_epoch`, `joining` and `leaving` booked for the epoch after, `created_epoch` and `last_active_epoch`. |
| **Membership** | One account per validator identity, seed `["member", identity]`. The member's hub and the id that hub had when it joined, the epoch the membership counts from, a pending leave or move with its target and epoch, and the cooldown. |

Every field, its type, size and offset, is in Appendix C.

The id makes a released code safe to register again at the same address.

**A hub has an id.** `release_hub` does not wait for the last member to leave, and a released code can be registered again at the same address. The new hub gets the next id, `hubs_registered + 1`, so a membership in the earlier hub, whose `hub_id` no longer matches, counts for no hub, and the counts of the new one stay exact.

## 11.4 Instructions

Table 11.2 The instructions of `hub_registry`. Every one but `initialize` emits its event by self CPI.

| Instruction | Signers | Effect | Event |
|---|---|---|---|
| `initialize(config)` | Upgrade authority | Creates Registry and FeeVault with the configuration | |
| `register_hub(code, location)` | Registrant | Creates the hub; the deposit into the hub account, the fee into the fee vault | `HubRegistered` |
| `join_hub()` | Identity, payer | Joins a hub from the next epoch | `MemberJoined` |
| `leave_hub()` | Identity | Leaves at the next epoch boundary; starts the cooldown | `MemberLeaving` |
| `move_hub()` | Identity | Moves to another hub at the next epoch boundary; starts the cooldown | `MemberMoving` |
| `release_hub()` | Anyone | Closes a hub inactive past its period; deposit and rent to the registrant | `HubReleased` |
| `sweep_fees()` | Anyone | Sends the fee vault's balance above rent to the fee destination | `FeesSwept` |

### 11.4.1 Configuration

The registry is configured once, by the program's upgrade authority, in `initialize`. The instruction reads the program's ProgramData account and requires its upgrade authority to be the signer, so the configuration cannot be set by whoever sends the first transaction after a deploy. The fee rate is at most 10,000 basis points, and the code length range lies within 1 to 8. There is no update instruction: a change ships as a program upgrade.

The deposit, the registration fee rate, the cooldown, the inactivity period and the member floor are configuration points set at deploy, read from the Registry account by every client. The live values are served by `GET /v1/registry/config`. The fee is computed from the deposit: `deposit_lamports × registration_fee_bps / 10,000`.

### 11.4.2 Registering a hub

`register_hub(code, location)` checks the code and the location and creates the hub at `["hub", code]`.

— **Code.** Capitals `A` to `Z` only, with a length within the registry's range: 3 to 8 on Acrux's configuration, an airport code such as FRA or a longer name. Because the code is the seed, a code that is taken fails at account creation.

— **Location.** An ECEF point in the WGS84 frame, whole metres, whose distance from the Earth's centre lies between 6,300 km and 6,400 km: a point near the surface. The CLI turns a latitude and longitude into that point on the ellipsoid at sea level (`acrux hub register FRA --latitude … --longitude …`).

— **Payment.** The registrant pays the deposit into the hub account, where it stays above the account's rent, and the fee into the fee vault, both by system transfers in the same instruction. The deposit returns at release; the fee is the protocol's share.

A new hub has no members. Its `last_active_epoch` starts at its creation epoch, so its inactivity period counts from registration: it has that many epochs to gather $n_{min}$ members.

### 11.4.3 Joining, leaving and moving

A validator joins by its identity. `join_hub` takes the identity as a signer and a separate payer, so an operator wallet can pay the membership's rent while the validator identity signs; both may be one key. The membership account is created on the identity's first join. Its `effective_epoch` is the next epoch: the join is booked on the hub as `joining`, and the hub counts the member from the boundary on. A first join starts no cooldown.

`leave_hub` and `move_hub` are accepted only when the membership is in effect, nothing is pending and the cooldown is over. Each books the change for the next epoch, records it on the departing hub as `leaving` (and on the target of a move as `joining`), and sets `cooldown_until_epoch` to the epoch the change takes effect plus `cooldown_epochs`. A change in effect is final from its epoch on, whether or not an instruction has written it into the membership yet; the next instruction on the membership settles it.

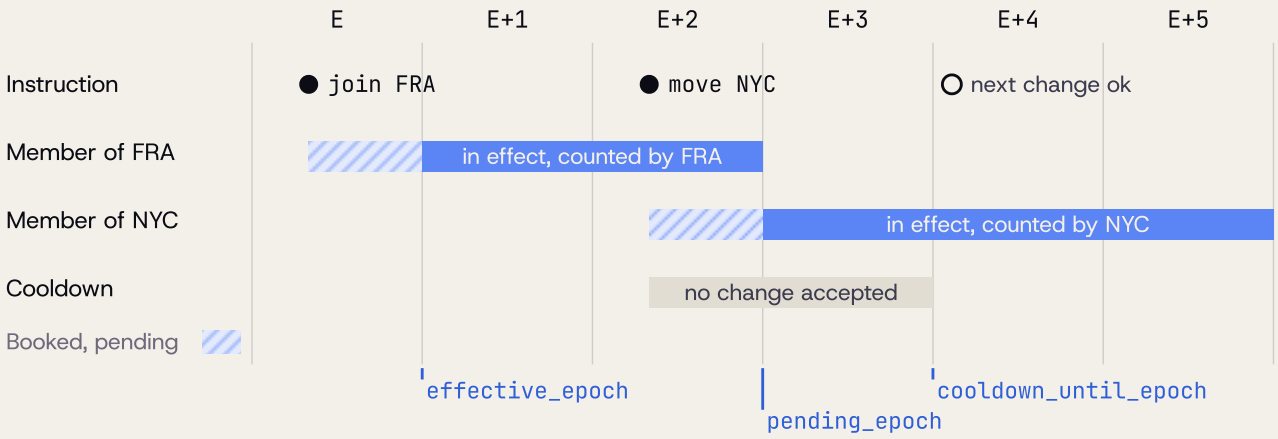



**Figure 11.2**

A membership across six epochs, with a cooldown of 1 epoch. A join sent in epoch E counts from E+1. A move sent in E+2 is booked (hatched) and in effect from E+3, the move's `pending_epoch`; the cooldown runs one epoch past it, so the next change is accepted from E+4.

### 11.4.4 Activity without a crank

A hub is active in an epoch when at least $n_{min}$ memberships are in effect in it. The program keeps that fact exact without anyone touching the hub each epoch. Joins and departures are booked as `joining` and `leaving` for the epoch after `counted_epoch`, so the hub's count in every epoch is known from its own account: `members` at `counted_epoch`, `members + joining - leaving` after it. Every instruction that touches a hub first rolls the counts forward to the current epoch (`Hub::roll`): if the count in effect reached $n_{min}$, `last_active_epoch` advances to the last epoch that had it. Between two instructions the count does not change, so no epoch is missed.

`release_hub` runs the same roll before it checks the period, so no one has to touch a hub to keep it alive or to make it releasable. A hub is releasable when the current epoch is past `last_active_epoch + inactivity_epochs`. Anyone may send it: the hub account closes, the deposit and the account's rent go to the registrant (checked against the hub by `has_one`), and the code is free. Memberships pointing at the released hub count for no hub; their identities leave or move as usual, and the departure is recorded on a hub only when its account still exists under the same id.

The paper's second floor, a hub stake of at least $\rho_{min}$ = 1% of total stake, is applied off chain by the schedule (Section 11.7), since the program does not read Solana stake. On chain, `last_active_epoch` tracks the member floor.

### 11.4.5 Fees

`sweep_fees` sends everything the fee vault holds above its rent to the registry's `fee_destination`. It is permissionless because the destination is fixed in the registry; it fails with `NothingToSweep` when there is nothing above rent. The fee is the protocol's fee share, and its use is told with the token (Chapter 14).

---

## 11.5 Flows

Three of the product's flows run on the registry (`PLAN.md`, Flows). Each can be done from the site, with a wallet connected on Solana, or with the `acrux hub` commands signed by the validator identity (Chapter 13).

### 11.5.1 F4 · Register a hub

1. Open `/hubs/register` and connect a wallet on Solana.
2. Enter a code of 3 to 8 capitals and pick the location on the map; it is stored as ECEF.
3. The page shows the deposit and the fee, read from the Registry account.
4. Sign `register_hub`. The hub appears as pending on `/hubs`.

Moves: the deposit from the registrant into the hub account, returned on release; the fee into the fee vault.

### 11.5.2 F5 · Join a hub

1. The operator runs `acrux hub join FRA --identity <keypair>`, or signs on `/hubs/register` with the identity in a wallet.
2. The membership is pending until the next epoch boundary.
3. At the boundary it takes effect; the member's proposer stake is computed from its stake.
4. The hub page shows it, and the schedule preview places it.

Leave and move work the same way, with the cooldown. Moves: the rent of the membership account.

### 11.5.3 F6 · Release an inactive hub

1. Anyone calls `release_hub` on a hub inactive past its period, from `/hubs/[code]` or with `acrux hub release <CODE>`.
2. The hub closes, the deposit and the rent return to the registrant, and the code is free.

Moves: the deposit back to the registrant.

---

## 11.6 The indexer

The API reads each chain's registry into Postgres for the site and for the schedule preview ( `backend/api/src/registry.ts` ), one indexer per chain. Every tick, two seconds by default ( `REGISTRY_POLL_SECONDS` ), it makes two calls: `getEpochInfo` and `getSignaturesForAddress` on the program with a limit of one. Only when the epoch or the program's newest signature has changed does it read the program's accounts with `getProgramAccounts`. It tells Registry, Hub and Membership apart by their 8-byte discriminators and decodes each by its Borsh layout.

- **Memberships in effect.** A membership counts for its hub from its `effective_epoch` until a pending leave or move reaches its `pending_epoch`, the same rule the program applies.
- **Location and region.** The ECEF point is turned into latitude and longitude on WGS84 (Bowring's method), and the latitude and longitude into the region SIMD-0675 keeps a run inside: North America, South America, Europe, Africa and Middle East, Asia Pacific or Oceania.
- **Stake and caps.** Each member's stake comes from the stake snapshot; the hub's members and stakes go through `acrux schedule`, the node's own binary, which returns each member's proposer stake under the $1/k$ cap.
- **Status.** A hub is `active` when it has at least $n_{\min}$ members in effect and a proposer stake of at least 1% of total stake; `inactive` once the epoch has reached its releasable epoch; `pending` otherwise. The read model also keeps what a hub still needs to become active, in members and stake.

Hubs and members are rewritten in one database transaction per change, so a reader never sees half an epoch. The API serves them as `GET /v1/hubs`, `GET /v1/hubs/{code}`, `GET /v1/registry/members/{identity}` and `GET /v1/registry/config`, the configuration read from the Registry account; `/hubs` and `/hubs/[code]` render them.

## 11.7 From the registry to Algorithm 3

On Acrux, the node reads the registry's accounts directly for its own schedule (Section 11.2). Algorithm 3 takes the active hubs $H$, the proposer stakes of their members and the hub stakes, and a PRNG seeded from the epoch (Chapter 6). The registry supplies the first two, at an epoch:

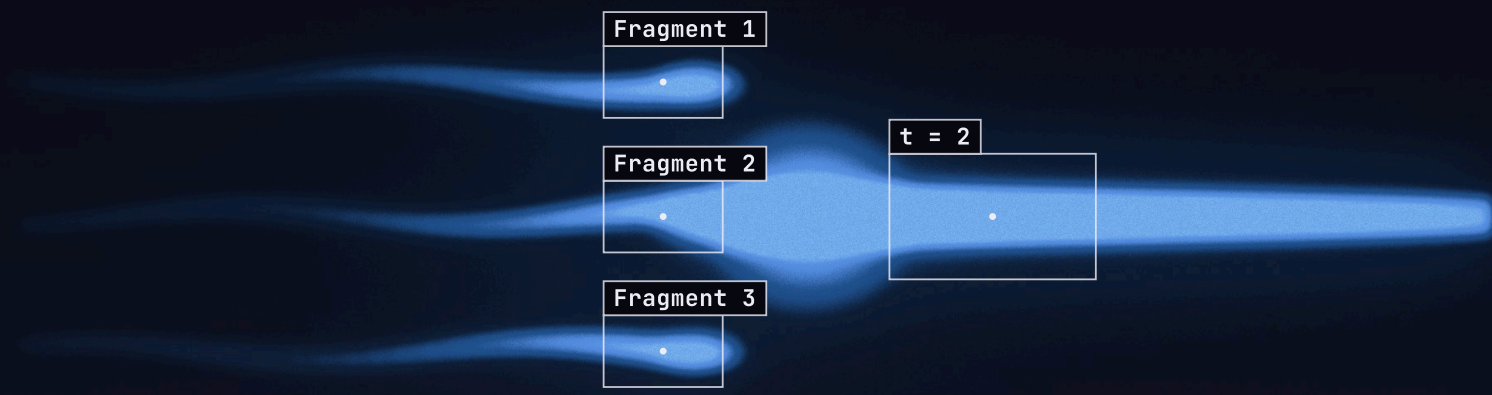
1. **Members.** For epoch $e$, every membership in effect at $e$, by the reading rules above. A change sent during an epoch takes effect at the next boundary, so the memberships of an epoch are fixed when it begins.
2. **Active hubs.** The hubs with at least $n_{\min}$ members in effect and a hub stake of at least $\rho_{\min}$ of total stake.
3. **Proposer stakes.** Each member's stake from the snapshot, capped so that no member holds more than $1/k$ of its hub's proposer stake; the hub stake is their sum.
4. **Regions.** Each hub's region, from its registered location, for the SIMD-0675 reordering that keeps runs of windows inside a region [12].

The API's schedule preview, `GET /v1/schedule`, runs exactly this through `acrux schedule`, the code path of the node's own schedule, with the seed `acrux-schedule-v1` and 96 windows per epoch. For each window it returns the hub, the first and last slot, the four consensus slots and the proposers with their indices and proposer stakes; for the epoch it returns each hub's share of windows and the regional runs, with $\alpha = 1/2$. The preview is what `/hubs` shows as the next epoch's schedule, so a validator that joins a hub sees its windows placed before its membership takes effect.

> **The registry turns hubs into accounts every validator reads alike, and the schedule turns those accounts into windows.**

```
await acrux.send(tx, { n: 3, t: 2 })
```

2-OF-3 FEE · 8,000 LAMPORTS




# 12

# The SDK

One call sends a transaction to n proposers of the target slot, split so that any t rebuild it and fewer read nothing.

`@acrux/sdk` is the client side of a leaderless slot. It reads a signed Solana transaction, picks the target slot and its proposers, splits the transaction in the client when the sender asks for a threshold, and follows it until its cblock is final. It runs in the browser and on Node 22.

## 12.1 Install and connect

The SDK ships as two private tarballs: the SDK itself and `@acrux/contract`, the package that holds every schema, type and raw client the SDK builds on. Both are packed into `backend/dist-pack/`; a project copies them into its `vendor/` folder and installs them together.

```
npm install ./vendor/acrux-contract-0.1.8.tgz \
            ./vendor/acrux-sdk-0.1.0.tgz
npm install @solana/web3.js
```

Table 12.1 The two packages a client installs.

| Package | Version | Holds |
|---|---|---|
| `@acrux/sdk` | 0.1.0 | `connect`, `send`, `tx.follow`, `slots.subscribe`, `seal`, `open`, `deriveHeader`, `headerBytes`, `AcruxError` |
| `@acrux/contract` | 0.1.8 | every zod schema and type of the gateway, the event stream and the HTTP API; `createGatewayClient`; `createClient` for the API |

The SDK depends only on the audited Noble primitives: `@noble/ciphers` for ChaCha20-Poly1305, `@noble/curves` for Ed25519 and `@noble/hashes` for SHA-256. `@solana/web3.js` builds and signs the transaction; `send` accepts its `Transaction`, its `VersionedTransaction`, or the serialized bytes of either.

```
import { connect } from '@acrux/sdk'

const acrux = connect('http://127.0.0.1:4930')
// the stream is read one port above: ws://127.0.0.1:4931
const other = connect(url, { wsUrl: 'wss://events.example' })
```

`connect` takes the gateway's JSON-RPC URL and returns a client. The event stream URL is derived from it: the same host, the port one above, `ws:` for `http:` and `wss:` for `https:`. A deployment that serves the stream on its own name passes `wsUrl`. The client holds `latestBlockhash`, `send`, `tx.follow`, `slots.subscribe`, and `gateway`, the contract's client for every JSON-RPC method and the raw event stream (Appendix B).

## 12.2 Sending a transaction

An Acrux transaction is a signed Solana transaction. Acrux Net executes in its own SVM, so the transaction carries the net's blockhash, not a Solana cluster's. Any blockhash of the last 1,200 slots, one minute, is accepted; `latestBlockhash()` returns the one from 8 slots before the latest executed slot, which every validator already holds.

```
const cb = ComputeBudgetProgram
const { blockhash } = await acrux.latestBlockhash()
const tx = new Transaction({
  feePayer: payer.publicKey,
  recentBlockhash: blockhash,
}).add(
  cb.setComputeUnitLimit({ units: 20_000 }),
  cb.setComputeUnitPrice({ microLamports: 100 }),
  SystemProgram.transfer({
    fromPubkey: payer.publicKey,
    toPubkey: to,
    lamports: 10_000,
  }),
)
tx.sign(payer)

const { id, targetSlot } = await acrux.send(tx)
await acrux.send(tx2, { n: 3, t: 2, signer: payer })
```

`send(tx, { n, t, signer })` takes three options. `n`, 1 to 3 and 1 by default, is how many of the target slot's $k = 3$ proposers receive the transaction; `t`, 1 to `n` and 1 by default, is how many of them must include it before anyone can read it; `signer` is the fee payer, a `Keypair` or a function `(message: Uint8Array) => Uint8Array | Promise<Uint8Array>`, required when `t` is 2 or 3. With `t` = 1 every receiver gets the signed transaction in the clear; with `t` of 2 or 3 each receiver gets one encrypted fragment, and the fee payer signs the public header the fragments carry.

Table 12.2 What each combination of `n` and `t` puts on the wire, and what it buys.

| Call | What is sent | What it buys |
|---|---|---|
| `send(tx)` | the signed transaction to one proposer | One proposer carries it in the clear. Cheapest. |
| `send(tx, { n: 2 })` | the signed transaction to two proposers | Two copies; either proposer lands it. Survives one drop. |
| `send(tx, { n: 3, t: 2, signer })` | three encrypted fragments | Split three ways; any two rebuild it, no one reads it alone. |
| `send(tx, { n: 3, t: 3, signer })` | three encrypted fragments | All three fragments needed. Most private. |

12.2.1

### What send does, step by step

Everything a proposer needs to order and schedule a transaction without reading it is derived from the signed bytes in the client, by the same rules the node applies to the same bytes. `send` runs these steps in `backend/sdk/src/index.ts`:

1. **Validate.** `1 ≤ t ≤ n ≤ 3`, and a `signer` when `t` > 1. Anything else is refused with `INVALID_INPUT` before a request leaves the client.
2. **Read the header.** `deriveHeader` (`transaction.ts`) parses the wire format, legacy or versioned, and builds the public `TxHeader`: `id`, the SHA-256 of the signed message in lowercase hex; `feePayer`, the first account key; `priorityFeePerCu` from a `SetComputeUnitPrice` instruction, 0 without one; `computeLimit` from `SetComputeUnitLimit`, or 200,000 CU per non-Compute-Budget instruction, at most 1,400,000; and `writableAccounts` by the message header's signer and read-only counts, with invoked programs demoted to read-only unless the upgradeable loader is present, as the runtime demotes them.
3. **Pick the slot.** The target is four slots after the latest started slot, `getRecentSlots({ limit: 1 })` plus 4: 200 ms ahead at $\delta_{\text{block}}$ = 50 ms, enough for the gateway to hand the transaction to the proposers before their block for that slot opens.
4. **Pick the receivers.** It reads the window holding the target slot, from `getCurrentWindow` or the next eight windows of `getSchedule`, and takes its proposers with status `online` first, then the rest, `n` of them by proposer index.
5. **Submit in the clear** when `t` = 1: one `submitTransaction` with the header, the signed bytes in base64 and the list of receivers. The gateway hands the transaction to each receiver for `header.targetSlot`.
6. **Sign the header** when `t` > 1: the fee payer signs `headerBytes(header)`, the header as compact JSON with its six fields in a fixed order. A `Keypair` signs with Ed25519 in the client; a function, such as a wallet adapter's `signMessage`, is called with the bytes. The signature is verified against `header.feePayer` before anything is sent.
7. **Seal and submit.** `seal` produces one `FragmentSubmission` per receiver, and `send` makes the `n` `submitFragment` calls in parallel.

`send` returns as soon as the gateway admits the transaction, with `{ id, targetSlot }`. Sending the same signed transaction again for the same slot is idempotent: the gateway answers with the same submission.

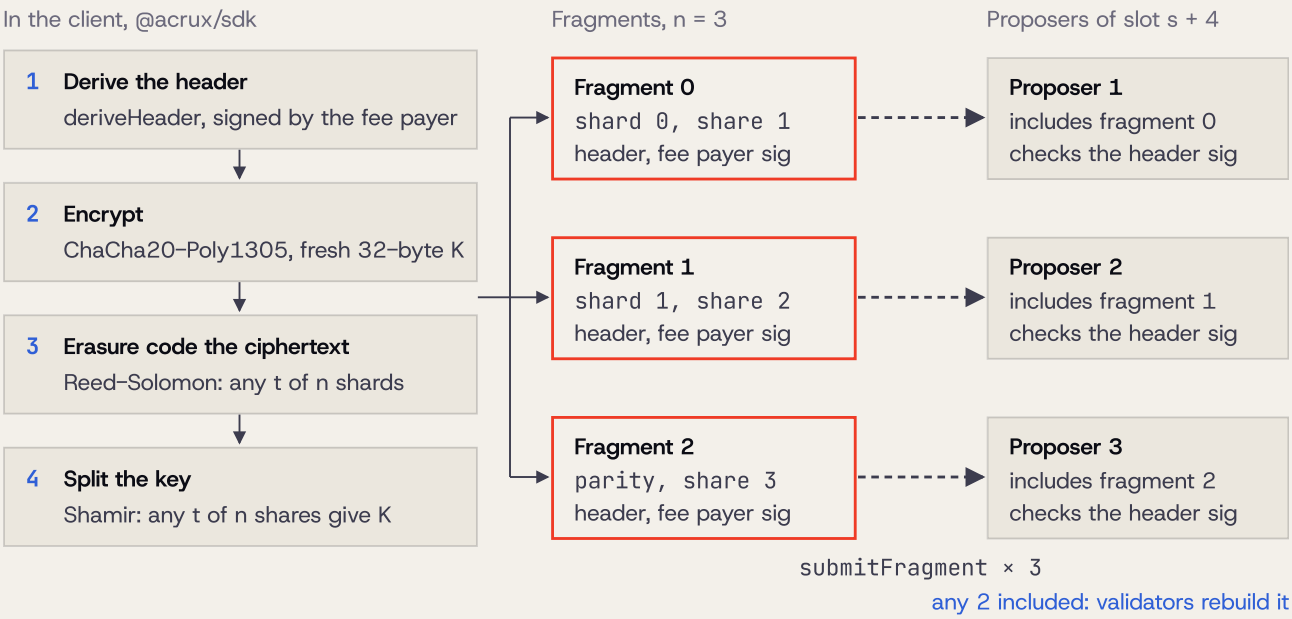



**Figure 12.1** The send path of `send(tx, { n: 3, t: 2, signer })`. The transaction is split in the client; each proposer of the target slot receives one fragment that it cannot read alone, checks the fee payer's signature over the public header, and includes it. Once two fragments are included in the same slot, every validator rebuilds and executes the transaction.

12.2.2

## The split in the client

`seal(bytes, header, headerSignature, t, receivers)` in `backend/sdk/src/threshold.ts` is the threshold encoder of the paper's Section 3.2 [1], byte-compatible with the node's `seal_for_receivers` and with the decoder in `acrux-execution/src/threshold.rs`. Its three layers follow the construction of Krawczyk [11]: encrypt once, erasure code the ciphertext, secret-share the key.

- **Encryption.** A fresh 32-byte key $K$ and a 12-byte nonce from `crypto.getRandomValues`; ChaCha20-Poly1305 over the signed transaction with `headerBytes(header)` as associated data. The header is therefore bound to the ciphertext: a fragment carried under a different header fails authentication when it is opened.
- **Erasure code.** The ciphertext is cut into `t` data shards of `⌈len / t⌉` bytes and extended with `n - t` parity shards by Reed-Solomon over GF($2^8$) reduced by 0x11d, with the systematic matrix of the `reed-solomon-erasure` crate: a Vandermonde matrix times the inverse of its top square. Row `i` makes shard `i`, so any `t` shards rebuild the ciphertext.
- **Key shares.** Each byte of $K$ is the constant term of its own random polynomial of degree `t - 1` over GF($2^8$) reduced by 0x11b, the node's Shamir field. Share `i` is every polynomial evaluated at `x = i + 1`; `t` shares give $K$ by Lagrange interpolation at 0, and fewer give nothing about it.

Table 12.3 One `FragmentSubmission` as `seal` writes it. `fragment` and `keyShare` are base64; the full wire layout is in Appendix D.

| Field | Bytes | Holds |
|---|---|---|
| `header` | JSON | the public `TxHeader`, the same in every fragment |
| `n, t, index, receiver` | integers | the split, this fragment's index from 0, and the proposer index it goes to |
| `fragment` | `01 ‖ nonce[12] ‖ len u32 LE ‖ shard` | version 1, the cipher nonce, the ciphertext length, then shard `index` |
| `keyShare` | `01 ‖ x ‖ share[32]` | version 1, the x coordinate `index + 1`, one share byte per key byte |
| `headerSignature` | base58 | the fee payer's Ed25519 signature over `headerBytes(header)` |

Vectors: `backend/sdk/test/vectors/rust-*.json`, written by the Rust encoder.

`open(fragments)` is the inverse, the way the node decodes: it takes any `t` distinct fragments of one seal, inverts the rows of the encoding matrix those fragments hold, rebuilds the ciphertext, interpolates the key and decrypts. The SDK's tests run both directions against fragments the Rust encoder wrote, for 2-of-2, 2-of-3 and 3-of-3: the header derived in TypeScript equals the node's, every `t`-subset of the Rust fragments opens in TypeScript, every `t`-subset of TypeScript fragments opens and parses under the contract's `FragmentSubmission`, the parity shard equals the crate's `galois_8` parity byte for byte, and a fragment opened under a changed header is refused.

```
import { deriveHeader, headerBytes, seal, open }
  from '@acrux/sdk'

const header = deriveHeader(bytes, targetSlot)
const sig = await signer(headerBytes(header))
// t = 2; receivers are proposer indices
const fragments = seal(bytes, header, sig, 2, [1, 2, 3])
const rebuilt = open(fragments.slice(1)) // any two
```

## 12.3 The lifecycle of a transaction

A transaction resolves through a fixed sequence of states, recorded by the gateway in its `Transaction` record and pushed on the event stream as `transaction.updated`. `tx.follow(id)` turns that stream into an async iterator.

```
for await (const u of acrux.tx.follow(id)) {
  console.log(u.state, `${u.included} of ${u.threshold}`)
}
// sent 0 of 2 · included 1 of 2 · included 2 of 2
// executed 2 of 2 · finalized 2 of 2
```

Table 12.4 The states `tx.follow` yields, in order. The last three end the loop.

| State | Arrives when |
|---|---|
| `sent` | the gateway admitted the transaction for its target slot |
| `included` | a proposer included a copy or a fragment in its block; once per inclusion, `included` counting up toward `threshold` (the record's `partly_included`) |
| `executed` | the transaction ran in the xblock of the slot where its `t`-th fragment was accepted |
| `finalized` | its cblock carries a fast or slow finalization certificate |
| `failed` | it executed and failed; its fees are charged |
| `expired` | fewer than `t` fragments were included within 8 slots of the first included fragment |

The iterator is exact about order and gaps. It opens a stream filtered to the transaction, `{ kind: 'transaction', id }`, and once the stream is open reads the gateway's record with `getTransaction`, so a state reached before the stream opened is not missed. Each record is ranked by its stage (sent 0, included 1, executed 2, a final state 3) and its inclusion count, and only a record that moves forward is yielded, so the same state reported by several validators arrives once and a late record never moves the trail back. An update carries `id`, `state`, `included`, `threshold` and the full contract `Transaction`: header, receivers, each inclusion with its slot and proposer, the readable and executed slot, the merged position, the result, program logs and the finalization certificate. Leaving the loop early with `break` or `return` closes the stream; a stream that closes under the loop raises `NET_UNAVAILABLE`.

### 12.3.1 Slots and the raw stream

```
const stop = acrux.slots.subscribe(slot =>
  console.log(slot.slot, slot.status, slot.agreedSet))
stop()

const sub = acrux.gateway.subscribe({ kind: 'all' }, e => {
  if (e.type === 'slot.agreed')
    console.log(e.slot, e.data.agreedSet)
})
sub.close()
```

`slots.subscribe` delivers every `Slot` twice: on `slot.started`, still pending, and on `slot.executed`, with its agreed set *P*, merged order, four-track schedule and state hash. Every validator reports each slot; the SDK keeps the

highest slot seen per stage and delivers only the first report of each. `acrux.gateway.subscribe` is the contract's raw client for any filter and every event type of Appendix B; it opens with a `stream.snapshot`, and on any `stream.resync` the subscriber replaces its state with the snapshot it carries. Reads at rest go through `acrux.gateway.call`, typed by the contract: `getCurrentWindow`, `getRecentSlots`, `getTransaction`, `getBalance` and the rest.

## 12.4 Typed errors

Every failure the SDK raises is one class, `AcruxError`, which is the contract's `ApiClientError` under the SDK's name. A refusal by the gateway and a refusal of the input by the SDK itself have the same shape, so a caller branches on one field.

```
import { AcruxError } from '@acrux/sdk'

try {
  await acrux.send(tx, { n: 3, t: 2, signer: payer })
} catch (error) {
  if (error instanceof AcruxError
      && error.code === 'TRANSACTION_EXPIRED') {
    // sign over a fresh blockhash and send again
  }
}
```

An `AcruxError` carries `code`, the stable `ErrorCode` to branch on; `status`, the HTTP status that code carries; `message`, a sentence that can be shown as it is; and `body.error`, the whole envelope with `retryable`, `stage` and, for invalid params, `fieldErrors` naming each field.

Table 12.5 Every error a send, a follow or an open can raise, and where it comes from.

| Code | Raised by | Cause | What to do |
|---|---|---|---|
| `INVALID_INPUT` | SDK | `n` or `t` outside 1 to 3, or `t` greater than `n` | fix the options |
| `INVALID_INPUT` | SDK | `t` of 2 or 3 without a `signer` | pass the fee payer |
| `INVALID_INPUT` | SDK | `signer` is not the transaction's fee payer | sign with the fee payer |
| `INVALID_INPUT` | SDK | the bytes are not a signed Solana transaction | serialize a signed transaction |
| `INVALID_INPUT` | SDK | `open` got fewer than `t` distinct fragments, or fragments that fail authentication | pass `t` fragments of one seal |
| `NET_UNAVAILABLE` | SDK | the event stream closed while `tx.follow` read it | follow again; `retryable` |
| `INVALID_INPUT` | gateway | a field failed validation, a receiver named twice, or the target more than 64 slots ahead | fix the input |
| `TRANSACTION_EXPIRED` | gateway | the target slot had started when the gateway admitted the send | sign over a fresh blockhash, send again |
| `CHAIN_REFUSED` | gateway | the node refused the transaction or fragment | `message` says why |
| `IDEMPOTENCY_CONFLICT` | gateway | the ID was sent before with a different header, `n` or `t`, or this fragment index with other bytes | send a new transaction |
| `NET_UNAVAILABLE` | gateway | the gateway cannot reach the net | retry; `retryable` is true |

`body.error.retryable` says whether the same call can succeed later. A send refused with `TRANSACTION_EXPIRED` is not retried as it is, since its target slot has passed: the transaction is signed again over the blockhash from `latestBlockhash()` and sent. The JSON-RPC codes behind these and every `ErrorCode` of the contract are in Appendix B.

## 12.5 What a send costs

Each proposer that includes a copy or a fragment charges the inclusion fee, 1,000 lamports, so `n` included copies cost `n` inclusion fees. The base fee, 5,000 lamports per signature, and the priority fee are charged once, when the transaction executes. The fee payer keeps a reserve $\rho$ of 0.001 ACRUX that a send cannot spend; `getBalance` returns `spendableLamports` above it. The fee rules are in Chapter 5.

The figures below were measured on live Acrux Net by `backend/sdk/test/cost.mjs`: ten sends of each kind, each a 0.001 ACRUX transfer with no priority fee through `@acrux/sdk`, medians from submit.

Table 12.6 Send cost and speed on Acrux Net. The fee is the 5,000-lamport base fee plus 1,000 lamports per included copy or fragment.

| Send | Fee | Submit to executed | Submit to finalized |
|---|---|---|---|
| Plain, to 1 proposer | 6,000 lamports | 339 ms | 580 ms |
| Plain, to all 3 proposers | 8,000 lamports | 342 ms | 585 ms |
| 2-of-3 threshold | 8,000 lamports | 339 ms | 585 ms |

> A 2-of-3 send costs what a plain send to the same three proposers costs, 2,000 lamports more than a plain send to one, and reaches executed and finalized as fast.

## 12.6 The contract types

`@acrux/contract` is the one source of every shape that crosses a wire in Acrux: zod schemas, from which the TypeScript types, the API's OpenAPI 3.1 document and the gateway's JSON-RPC and event descriptions are generated. Every response and event the SDK hands back has parsed under these schemas. The SDK re-exports the types a client touches.

Table 12.7 The contract types and clients a client of the SDK uses. Every field of every type is in Appendix B.

| Export | Is |
|---|---|
| `TxHeader` | `id`, `feePayer`, `priorityFeePerCu`, `computeLimit`, `writableAccounts`, `targetSlot`: what a proposer and the merge see of a sealed transaction |
| `FragmentSubmission` | the params of `submitFragment`, one per receiver |
| `Transaction` | the trail: state, kind, `n`, `t`, receivers, inclusions, readable and executed slot, merged position, result, logs, failure, expiry, finalization |
| `Slot` | one slot whole: window, blocks, attestations, agreed set, skip, merged order, schedule, state hash, cblock |
| `ErrorCode` | the union of every stable error code of every Acrux service |
| `ApiClientError` | the error class, exported by the SDK as `AcruxError` |
| `createGatewayClient` | the typed JSON-RPC and event stream client, `acrux.gateway` |
| `createClient` | the typed client of the HTTP API: hubs, schedule, faucet, Fault Lab, facts, spec |

`npm test` in `backend/sdk` runs the seal and open round trips against the Rust vectors, and `npm run e2e` sends a plain and a 2-of-3 transfer on a running Acrux Net and follows each to finalized. The same send from a terminal is `acrux send` (Section 13.3.3).




13

# Operating a node and a hub

Run a three-hub net with one command, follow Acrux Net and re-execute every finalized slot, register a hub and join it with a validator identity.

One binary, `acrux`, runs a whole New Constellation net on one machine, follows a running net and re-executes it slot by slot, and signs a validator into a hub on the Hub Registry. This chapter is the operator's path through it: what changes for a validator, how to run, configure and watch a node, and how to register and join a hub.

## 13.1 What changes for a validator

Under New Constellation a validator no longer waits for its turn as the single leader of a slot. It proposes as one of $k = 3$ co-located proposers of its hub, for windows of $w = 16$ slots, and in every slot, proposer or not, it validates, merges, executes and votes like every other validator. Four things change in how it is run.

**Hubs.** A validator registers its location and joins a hub: a group of validators within $\delta_{\text{local}} = 10$ ms of each other, named by a code such as FRA. A hub is active with at least $n_{\min} = 4$ members and $\rho_{\min} = 1\%$ of total stake. A validator in no active hub only votes. Joining is a Solana transaction signed by the validator identity, in effect from the next epoch boundary (Section 13.7).

**Proposer stake.** Each member's stake is capped so that no member holds more than $1/k$ of its hub's proposer stake; the members at the cap propose in every window of the hub, and stake above it still counts for voting. Windows go to hubs by hub stake with probability $\alpha = 1/2$ and uniformly otherwise, so stake in a small hub earns more proposer roles per unit than stake in a large one (Chapter 6).

**Co-location.** The commit exchange closes $2\delta_{\text{local}} = 20$ ms after a proposer's `start[s]`. A co-proposer whose commitment misses it is attested late and its block is dropped from the slot, so the machine has to sit where its hub is. `acrux-node` runs at high process priority with raised thread priority so that a busy host does not delay a commitment past the window.

**Every slot.** A proposer builds and encrypts a block for $\delta_{\text{block}} = 50$ ms, commits, attests with its key, then validates, merges and executes the agreed blocks; votes go to cblocks of $\kappa = 4$ slots through Alpenglow (Chapter 3, Chapter 4).

## 13.2 Install

`acrux` is one binary: the proposer and validator, Rotor, Alpenglow on cblocks, threshold transactions and execution in an embedded SVM, with the gateway that serves JSON-RPC, the event stream and metrics. It builds from the Rust workspace in `backend/node`, whose six crates are described in Chapter 8.

A build needs Rust with Cargo (edition 2021, built with `--locked`), the MSVC build tools on Windows, and for the first build of the vendored static OpenSSL a complete Perl distribution set as `PERL`; later builds reuse that OpenSSL.

```
# Linux, as backend/deploy/Dockerfile builds it
cargo build --release -p acrux-node
# Windows: fixed target dir, reused static OpenSSL
powershell -ExecutionPolicy Bypass `
  -File scripts/build.ps1 --release

acrux net up --run-seconds 30   # check the build
```

The binary is `target/release/acrux` (`acrux.exe` on Windows). Thirty seconds of a twelve-validator cluster end with a summary line and a report, which is the check that the build runs.

## 13.3 A local three-hub net

`acrux net up` runs a whole New Constellation net on one machine: twelve validators in three hubs, FRA, NYC and TYO, four in each, every validator its own actor with its own key, message state, executor and Votor. They exchange signed messages over loopback QUIC, one endpoint per validator, with the delays of their hubs' geography applied on each link: the cities' distances at two thirds of the speed of light in fibre, plus 8 ms. Acrux Net runs from the same command.

```
acrux net up
acrux cluster up --hubs FRA,NYC,TYO
```

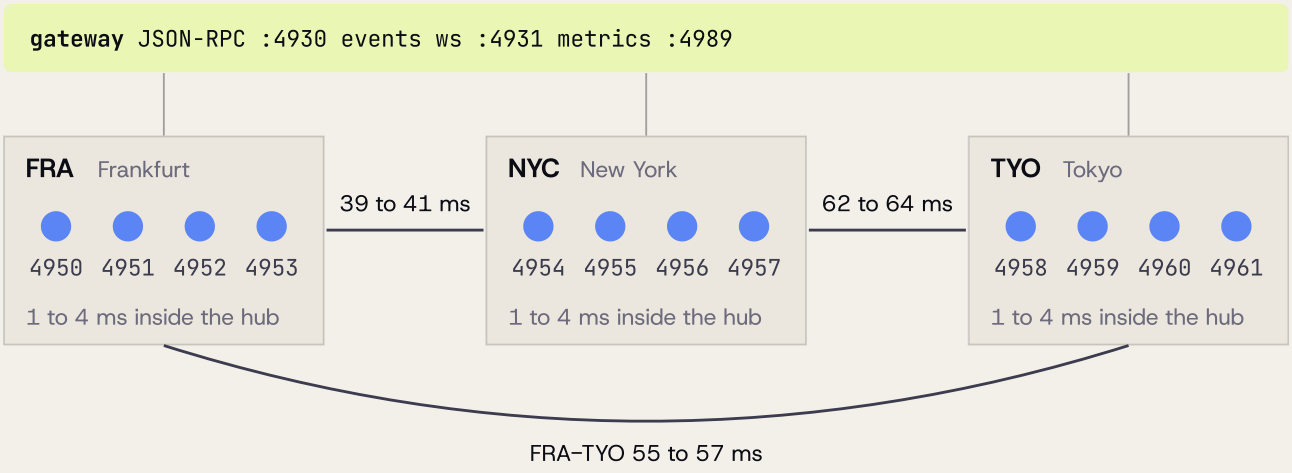



**Figure 13.1** The net `acrux net up` starts with its defaults: twelve validators, one QUIC endpoint each from `--base-port` 4950, in three hubs, with one-way delays on the wire inside each hub and between hubs (FRA–NYC 39 to 41 ms, NYC–TYO 62 to 64 ms); the gateway serves the whole net on 127.0.0.1.

The twelve validators hold equal stake; Algorithm 3 draws three proposers per window; slots are 50 ms, 16 to a window and 4 to a cblock. A 60% certificate needs eight of the twelve validators, so every cblock is decided by at least two hubs. The genesis carries the demo pool program, its AXA and AXB mints and vaults, and the faucet, and each start is a fresh genesis. Point the SDK or `/net` at the gateway and the cluster is a net like any other. `acrux cluster up` starts the same net named by its hubs, the three genesis hubs, with the four port flags and `--store` (default `target/runs/cluster`).

### 13.3.1 Bounded runs, a crashed proposer, extra programs

```
acrux net up --run-seconds 30 --traffic-rate 50
acrux net up --crash-window 2 --crash-proposer 3 \
             --run-seconds 30
acrux net up --program
<ADDRESS>=../chain/target/deploy/demo_pool.so
```

`--run-seconds` stops the run after that many seconds; without it the net runs until Ctrl+C. `--crash-window W --crash-proposer Q` silences proposer `Q` of window `W` from the window's start: the window continues on $P = \{1, 2\}$, and each validator holds its first vote of the window until the attestation deadline, the case the Fault Lab's Crash scenario runs (Chapter 10). Each `--program ADDRESS=PATH` loads an SBF program into every validator's SVM before the first slot, and repeats for more programs.

### 13.3.2 The report and the store

On stop the node closes its transport, writes `last-run.json` and `profile.json` into its store directory and prints one JSON line: `report`, the path of `last-run.json`; `finalizedCblocks`; `skips`; and `slotPacingUs` with its count, p50, p99 and maximum.

The report holds slot pacing inside windows and the gap at each window boundary, the handover gap per hub link keyed like `FRA-NYC`, commit spread, cblock finalization time, finalized and skipped cblocks, the heights whose state hashes were compared across validators and how many differed, the largest execution skew and lag between validators, each validator's final state hash and last executed slot, and the transfers the traffic generator sent.

`--store` names the run's directory, `target/runs/net` by default, relative to the working directory. It holds `genesis.json` (the run's configuration, seed, hubs and their members' identities), `net-<start>.redb` (slot, cblock and transaction records, which reads of evicted slots fall back to), and the two files written on stop. Giving each run its own directory keeps its report: `--store target/runs/crash-30s`. The figures of the 600–second run of Acrux Net come from exactly this report (Chapter 9).

### 13.3.3 Sending from a terminal

```
acrux keygen --out key.json
acrux send --keypair key.json --to <address> \
           --lamports 1000000 --n 3 --t 2
```

`acrux keygen` writes a Solana keypair file and prints its address. `acrux send` builds a transfer with a 20,000 CU limit and a priority fee of `--priority` micro-lamports per CU (1,000 by default) over the net's latest blockhash, for the slot four after the latest started one, and sends it to proposers 1 to `n`: in the clear with `t` = 1, as sealed fragments otherwise, through the same encoder the SDK uses. It prints `sent <id> for slot <slot>`, then each state until `finalized`, `failed` or `expired`.

## 13.4 Configuration

`acrux net up` reads its settings from flags, from the environment, and from a `.env` file in its working directory. A flag wins over the environment, the environment over `.env`, and a setting given nowhere takes its default. The `.env` file holds one `KEY=value` per line; blank lines and lines starting with `#` are skipped, there is no quoting or shell expansion, and a variable already set in the process is kept.

Table 13.1 The flags of `acrux net up`, with their environment variables and defaults.

| Flag | Environment | Default | Meaning |
|---|---|---|---|
| `--gateway-port` | `ACRUX_GATEWAY_PORT` | 4930 | JSON-RPC, on 127.0.0.1 |
| `--ws-port` | `ACRUX_WS_PORT` | 4931 | the event stream, on 127.0.0.1 |
| `--base-port` | `ACRUX_BASE_PORT` | 4950 | the first of twelve node-to-node QUIC ports, `base` to `base + 11` |
| `--metrics-port` | `ACRUX_METRICS_PORT` | 4989 | metrics, health and the faucet's private grant route |
| `--traffic-rate` | `ACRUX_TRAFFIC_RATE` | 50 | generated transfers per second; 0 for none |
| `--run-seconds` | | none | stop after this many seconds and write the report |
| `--crash-window` | | none | the window in which one proposer is silent from its start |
| `--crash-proposer` | | 3 | that window's silent proposer, 1, 2 or 3 |
| `--program ADDRESS=PATH` | | none | an SBF program loaded before the first slot; repeatable |
| `--genesis` | | `../chain/genesis/acrux-net.json` | genesis accounts and programs: the demo pool, its mints, the faucet |

| Flag | Environment | Default | Meaning |
|---|---|---|---|
| `--store` | | `target/runs/net` | the run's directory |

Table 13.2 Settings read from the environment or `.env` only. Those marked positive must be greater than zero.

| Environment | Default | Meaning |
|---|---|---|
| `ACRUX_BCU` | 2,000,000 | compute units per track per slot, $b_{cu}$; positive |
| `ACRUX_BASE_FEE` | 5,000 | lamports per signature, half burned |
| `ACRUX_INCLUSION_FEE` | 1,000 | lamports per included copy or fragment, to the including proposer |
| `ACRUX_RESERVE` | 1,000,000 | the fee-payer reserve $\rho$ in lamports, 0.001 ACRUX; positive |
| `ACRUX_FRAGMENT_EXPIRY` | 8 | slots after the first included fragment before an incomplete threshold transaction expires; positive |
| `ACRUX_RETAIN_SLOTS` | 24,000 | slots of history the store keeps, 20 minutes |
| `ACRUX_BIND` | `127.0.0.1` | the address the gateway binds; 0.0.0.0 in a container |
| `ACRUX_CORS_ORIGINS` | | browser origins the gateway serves |
| `RUST_LOG` | `warn` | the log filter in `tracing` syntax: `info`, `acrux_node=debug` |

The protocol's own parameters are compiled in: $k$, $\kappa$, $w$, $\delta_{local}$, $\delta_{block}$, $\delta_{global}$, $\Delta_{timeout}$, $m$, $\alpha$ and the 64-shred, 32-to-rebuild slice coding. Their values are in Appendix A, and the gateway serves them in `getSnapshot`'s `parameters`. By default the gateway answers browser requests from the site's own origins on 127.0.0.1 and localhost, ports 4910 and 4911; another origin is refused with `ORIGIN_NOT_ALLOWED`, and a request with no `Origin` header, from a script or a server, is served. `acrux net up` exits non-zero on a setting out of range: a zero `ACRUX_BCU`, `ACRUX_RESERVE` or `ACRUX_FRAGMENT_EXPIRY`, a `--crash-proposer` outside 1 to 3, a malformed `.env` line, or a port already in use.

## 13.5 Telemetry

A running node reports on four surfaces: Prometheus metrics and health on the metrics port, structured JSON logs, the full protocol event stream, and the run report when it stops. `GET http://127.0.0.1:4989/metrics` serves the Prometheus text format, version 0.0.4.

Table 13.3 The Prometheus metrics of `acrux-node`. The same counters, with the net's status, come from `getCounters` and the `net.counters` event.

| Metric | Type | Meaning |
|---|---|---|
| `acrux_slots_total` | counter | slots started |
| `acrux_finalized_cblocks_total` | counter | cblocks finalized |
| `acrux_skips_total` | counter | cblocks skipped |
| `acrux_commit_spread_microseconds` | gauge | average spread from the first to the last commitment of a slot, µs |

```
# prometheus.yml
scrape_configs:
  - job_name: acrux
    static_configs:
      - targets: ['127.0.0.1:4989']
```

`GET /health`, on the metrics port and on the gateway port, returns `status` (`ready` once a window is producing, `waiting` before), `contractVersion`, the `@acrux/contract` version the gateway speaks, and `missingConfiguration`, the settings the node needs and lacks. Logs are JSON, one object per line, through `tracing`, filtered by `RUST_LOG`. Everything the node does in a slot is an event on `ws://127.0.0.1:4931`: each slice, commitment and arrival, attestation, the agreed set, the merge, the four-track schedule, execution, every vote and certificate; an event one node emitted carries that node's identity in `node` (Appendix B). The rolling figures of the last ten minutes, measured on the node clock (pacing, handover per hub link with its floor, commit spread, attestation arrival, cblock finalization), are served by `getFacts` and by the API's `GET /v1/facts`.

## 13.6 Following Acrux Net

`acrux node follow` runs a validator's execution beside a net without being one of its validators. It reads the net through its gateway, takes nothing on trust that it can recompute, and re-executes every finalized slot on its own SVM: the signatures are checked against the window's proposer identities, the agreed set is recomputed by the paper's own rule, the blocks are decrypted with the keys the attestations release, merged and executed, and the state hash that results must be the net's.

```
acrux node follow http://127.0.0.1:4930 --genesis
../chain/genesis/acrux-net.json

from cblock <c>: <n> accounts loaded, state <hash> matches
slot <s> re-executed  agreed [1,2,3]  state <hash> matches
checked 1
```

**The starting state.** The follower asks the gateway for `getExecutionSnapshot`: the executed state of the latest finalized cblock and the threshold fragment cache at that point. It loads the programs named in the net's genesis (`--genesis`, by default `../chain/genesis/acrux-net.json`), loads the snapshot's accounts into its own executor, and checks that the loaded state hashes to that cblock's state hash on the net. From there it follows every later finalized cblock; a finalized cblock finalizes its ancestors, so the follower walks back through the parents to the last cblock it ran and executes the chain forward.

For each slot of each such cblock, in slot order, it does what Algorithm 2's TryExecute does on a validator (`Replay::apply` in `backend/node/crates/acrux-node/src/cli.rs`):

1. **Every attestation is its proposer's.** Each attestation is rebuilt from its fields and its Ed25519 signature checked against the identity of the window's proposer with that index (the signed bytes are in Appendix D).
2. **The agreed set follows from the attestations.** `Agreed(s)`, the strict-majority rule over agreeing vectors $\pi$, is run on the checked attestations, and its result must equal the slot's `agreedSet`.
3. **The blocks open under the released keys.** The slot's blocks come from `getSlotBlocks`. Each agreed block is decrypted with the key from its proposer's attestation; the key must hash to the block's key commitment, the block must end in its final slice, and its commitment must be the one the co-proposers attested in $\pi$.
4. **The slot executes as the validators executed it.** Duplicates are removed, threshold fragments are checked against their signed headers and receivers and decoded at the t-th fragment, the transactions are merged and selected by Algorithm 4 with the fees and the reserve rule (Chapter 5), and the xblock runs on the follower's SVM.
5. **The state hash matches.** The resulting state hash must equal the slot's state hash on the net, and on the fourth slot of each cblock the cblock must carry a fast or slow finalization certificate whose stake share reaches its threshold, 80% or 60%.

It polls the gateway every 200 ms for newly finalized cblocks and prints each slot with its agreed set $P$, the first 12 hex digits of the state hash and the running count. In the check the backend ran against Acrux Net, 704 consecutive slots were re-executed with matching state hashes, 2-of-3 transfers among them. A slot that fails a check stops the command with the slot number and its reason.

Table 13.4 The reasons `acrux node follow` stops on.

| Reason | What failed |
|---|---|
| `the starting state does not hash to cblock …'s state hash` | the snapshot's state is not the net's state at that cblock |
| `attestation by a non-proposer` | an attestation's index names no proposer of the window |
| `attestation signature` | an attestation's signature does not verify under its proposer's identity |
| `the attestations do not agree` | the attestations yield no agreed set |
| `agreed set … differs from the attestations' …` | the slot claims a different *P* than its attestations give |
| `released key does not match the block's key commitment` | a key from an attestation does not open its block |
| `block … does not match its attested commitment` | a block is not the one its co-proposers attested |
| `signed header or target mismatch, fragment carried by the wrong proposer` | a threshold fragment does not belong where it was carried |
| `re-executed state … differs from the net's …` | the follower's execution of the slot gives a different state hash |
| `no finalization certificate` | a cblock's last slot shows no finalization certificate at its threshold |
| `cblock … does not extend cblock …` | a finalized cblock's chain does not lead back to the last one run |

## 13.7 Registering and joining a hub

A validator takes part in New Constellation's schedule through a hub. The `acrux hub` commands send the Hub Registry's instructions (Chapter 11), signed and paid by the validator identity, and every change takes effect at the next epoch boundary of the chain it was sent to. The registry runs on both chains: on Acrux, where its hubs and memberships drive the proposer schedule, and on Solana mainnet; `--rpc` names the chain, Acrux through the Solana-compatible JSON-RPC on its own endpoint or a Solana RPC. The identity is a Solana keypair file, `acrux keygen --out key.json` writes a new one, and it holds the chain's native token (ACRUX on Acrux, SOL on Solana) on the chain named by `--rpc` for the membership's rent and the fees; a registration also pays the deposit and the registration fee.

Every `acrux hub` command takes `--identity`, the keypair that signs and pays (default `acrux-key.json`), `--rpc`, the Solana JSON-RPC it sends through (default the local validator, `http://127.0.0.1:4990`), and `--program`, the Hub Registry program id (default `ABjgHL8TcjxUjx6vfwsVQyvxiXR6pFgMkeESgwK6CbDm`). Each prints `<instruction> confirmed: <signature>` once its transaction is confirmed, waiting up to 30 s. The full command reference is in Appendix B.5.

### 13.7.1 Joining (F5)

```
acrux hub join FRA --identity key.json
join_hub confirmed: <signature>
```

Every registered hub, its location and members are on `/hubs` and from `GET /v1/hubs`; a validator joins the hub whose members it reaches within $\delta_{\text{local}}$. `join_hub` creates the membership on the identity's first join, with the identity paying its rent; the membership counts from the next epoch, and a first join starts no cooldown. From that epoch it counts toward the hub's member floor of 4, it shows at `GET /v1/registry/members/{identity}`, and once the hub is active the proposer schedule draws its windows by stake; the upcoming windows are on `GET /v1/schedule`.

### 13.7.2 Registering (F4)

```
acrux hub register AMS --latitude 52.31 --longitude 4.76 \
                       --identity key.json
```

The code is 3 to 8 capitals `A` to `Z`, not yet taken. The latitude and longitude, in degrees, become an ECEF point on the WGS84 ellipsoid at sea level, in whole metres, which the program accepts between 6,300 km and 6,400 km from the Earth's centre. The registrant pays the deposit, held in the hub account, and the registration fee, a share of the deposit in basis points, into the registry's fee vault; both are served by `GET /v1/registry/config` before signing. The registrant then joins its hub like any member, and a new hub has the registry's inactivity period, counted from its creation epoch, to gather its four members.

### 13.7.3 Leaving, moving and the cooldown

`acrux hub leave FRA` ends the membership at the next epoch boundary; `acrux hub move FRA AMS` makes the identity a member of the second hub from the next boundary. Either is accepted once the membership is in effect, nothing else is pending and the cooldown has ended, and either starts a cooldown that runs for the registry's cooldown epochs after the change takes effect: `cooldown_until_epoch` is the effective epoch plus `cooldown_epochs`.

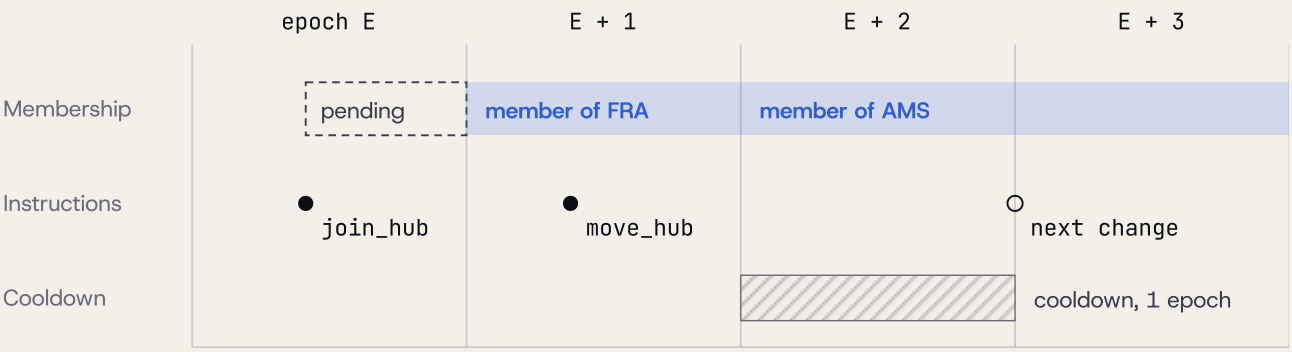



**Figure 13.2** One membership with the local registry's cooldown of one epoch. A join sent in epoch E counts from E + 1 and starts no cooldown; a move sent in E + 1 is in effect from E + 2 and starts the cooldown; the next leave or move is accepted from E + 3.

13.7.4

### Releasing a hub

`acrux hub release AMS` sends `release_hub` for a hub that has stayed below its member floor past the inactivity period: its epoch is past `last_active_epoch + inactivity_epochs`. Anyone can send it; `--identity` only pays the fee. The deposit and the account's rent return to the registrant, and the code is free to register again under a new hub id.

On a local Solana validator, `backend/chain/scripts/localnet.ps1` deploys the registry with a 1 SOL deposit, a registration fee of 500 bps, a cooldown of 1 epoch, an inactivity period of 4 epochs, $n_{min}$ of 4 and 30-second epochs, then registers FRA, NYC and TYO and joins Acrux Net's twelve validators to them.

A refused instruction comes back as `custom program error: 0x…` with an Anchor code: among them 6004 `AlreadyMember`, 6008 `MembershipPending`, 6009 `ChangePending`, 6010 `CooldownActive` and 6011 `HubNotInactive`. Every code, name and message is in Appendix C. A validator of Acrux Net's own genesis hubs gets its identity keypair from `acrux identities --out <dir>`, which writes the twelve identities derived from `ACRUX_IDENTITY_SEED` and prints each with its hub.

## 13.8 Ports and the public host

Table 13.5 Every port of a local Acrux deployment, all bound to 127.0.0.1.

| Service | Port | Serves |
|---|---|---|
| Acrux Net gateway | 4930 | JSON-RPC, /health |
| Acrux Net event stream | 4931 | WebSocket |
| Acrux Net node-to-node | 4950 to 4961 | QUIC, one endpoint per validator |
| Acrux Net metrics | 4989 | /metrics, /health, the faucet's private grant route |
| API | 4920 | the HTTP API |
| Postgres | 4921 | the API's read model |
| Local Solana validator | 4990 to 4993 | RPC, WebSocket, faucet, gossip; the Hub Registry |
| Fault Lab sandboxes | 5016 to 5027 | QUIC of one lab run, started by the API |
| Site | 4910, 4911 | development and production |

Acrux Net itself runs on one dedicated server, from backend/deploy/host/, as four containers: acrux-net (the twelve validators and the gateway), acrux-api (the API, the registry read model, the faucet and the Fault Lab sandboxes), acrux-postgres, and acrux-caddy, which terminates TLS for three public names: rpc. for JSON-RPC, events. for the event stream and api. for the API. Only ports 22, 80 and 443 are open; the node's QUIC, its metrics and the API are reachable only inside Docker. Every start is a new genesis from the host's genesis file with the same twelve validator identities, which derive from a secret seed kept on the host.

Table 13.6 The public limits of Acrux Net, per client address as Caddy appends it to X-Forwarded-For.

| Limit | Default | Variable |
|---|---|---|
| JSON-RPC reads | 50 a second, bursts of 100 | ACRUX_RPC_RATE |
| Transaction and fragment submits | 5 a second, bursts of 20 | ACRUX_SUBMIT_RATE |
| Event-stream connections | 8 | ACRUX_WS_PER_IP |
| Faucet grants | 3 per key and 10 per IP per 24 h, 2 test ACRUX plus 100 AXA and 100 AXB each | FAUCET_* |
| Fault Lab runs | 6 per IP an hour, at most 10 waiting or running | LAB_RUNS_PER_IP_HOUR, LAB_QUEUE_MAX |

**SUPPLY 1,000,000,000 · 9 DECIMALS**
GAS AND STAKE ON ACRUX · FAIR LAUNCH ON SOLANA
HALF OF EVERY BASE FEE BURNED



14

# ACRUX

The chain’s native gas and stake. The whole supply goes to a fair launch on Solana, and the Hub Registry’s fees buy ACRUX on the market and burn it.

ACRUX is the native token of the Acrux chain: every transaction pays its gas in it and every validator stakes it. Its whole supply goes to a fair launch on Solana, and the Hub Registry's fee buys ACRUX on the market and burns it.

## 14.1 Gas and stake

On Acrux, ACRUX is what SOL is on Solana. It has 9 decimals, and the chain counts it in lamports: 1 ACRUX is 1,000,000,000 lamports, and every amount in this paper given in lamports on Acrux is an amount of ACRUX.

Table 14.1 ACRUX on the chain.

| Role | What ACRUX does | Amount |
|---|---|---|
| Base fee | paid once per signature when a transaction is selected; half burned, half to the proposers that carried it | 5,000 lamports |
| Inclusion fee | paid per included copy or fragment to the proposer whose block carries it | 1,000 lamports |
| Priority fee | set by the sender per compute unit; orders the merged slot; to the proposers that carried it | per CU |
| Fee-payer reserve | kept by every fee payer so that its inclusion fees can always be paid (Chapter 5) | 0.001 ACRUX |
| Stake | held by validators; a hub's stake decides its share of windows in Algorithm 3, and a member's proposer stake is capped at $1/k$ of its hub (Chapter 6) | |
| Faucet | grants test ACRUX for sending on Acrux Net, with 100 AXA and 100 AXB | 2 ACRUX |

Every transaction on the chain therefore burns ACRUX: half of each base fee leaves the supply when the transaction is selected, whatever the transaction does.

## 14.2 Supply and launch

Table 14.2 The token.

| | |
|---|---|
| Symbol | `ACRUX` |
| Supply | 1,000,000,000 |
| Launch | fair launch on Solana: the whole supply goes to the public launch |
| Developer's allocation | the developer's own first buy in that launch, on the same terms as every buyer |

The supply is fixed at 1,000,000,000 ACRUX, and all of it goes to the public launch, on Solana. From there ACRUX comes onto Acrux as the chain's own currency, its gas and its stake. There is no allocation set aside before the launch, for the developer or anyone else: the developer's allocation is the developer's own first buy in the launch, made on the same terms as every other buyer. The size of that first buy is a configuration point of the launch.

## 14.3 Where the value comes from

The Hub Registry is the program where hubs are registered and joined, on Acrux, where its hubs drive the chain's schedule, and on Solana mainnet (Chapter 11). Registering a hub is the one action in it that pays the protocol, in the native token of the chain it runs on: ACRUX on Acrux, SOL on Solana. `register_hub` moves two amounts from the registrant:

- **The deposit,** `deposit_lamports`, held in the hub account above its rent. It returns to the registrant when the hub is released by `release_hub`, together with the account's rent.
- **The registration fee,** `deposit_lamports × registration_fee_bps / 10,000`, paid into the registry's `FeeVault`. The fee rate, in basis points of the deposit and at most 10,000, is set once in the registry's configuration by the program's upgrade authority. On Solana mainnet it is 500 bps of a 1 SOL deposit (Table C.1).

The `FeeVault` is a program-derived address, `["fee_vault"]`, that no key signs for. Fees sit in its lamports above its rent, and the only instruction that moves them is `sweep_fees`: anyone can call it, and it sends everything above rent to the registry's `fee_destination`, which is fixed in the `Registry` account, emitting `FeesSwept` with the amount. Every registration's fee is on chain in its `HubRegistered` event as `fee_lamports`.

## 14.4 Buy and burn

The registry's whole fee share buys ACRUX on the market, and every ACRUX bought is burned. The buy and the burn run through a crank on the fee vault, which ships as an upgrade of the registry once the ACRUX mint exists. From then on the path of a registration fee is fixed end to end: it enters the vault with the registration, leaves it only through the crank, and ends as ACRUX removed from the supply.

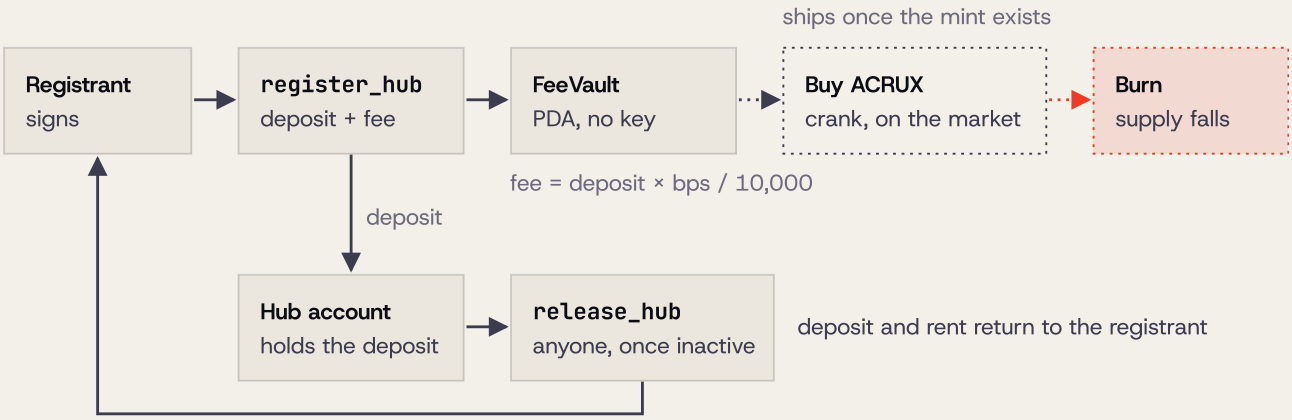



Figure 14.1 The flow of value through the Hub Registry. The registration fee enters the program-owned fee vault and leaves it through the crank that buys ACRUX on the market and burns it (dotted: the upgrade that ships once the mint exists). The deposit stays in the hub account and returns to the registrant on release.

The deposit beside the fee is the registrant's stake in the hub it names; it goes back in full when the hub is released.

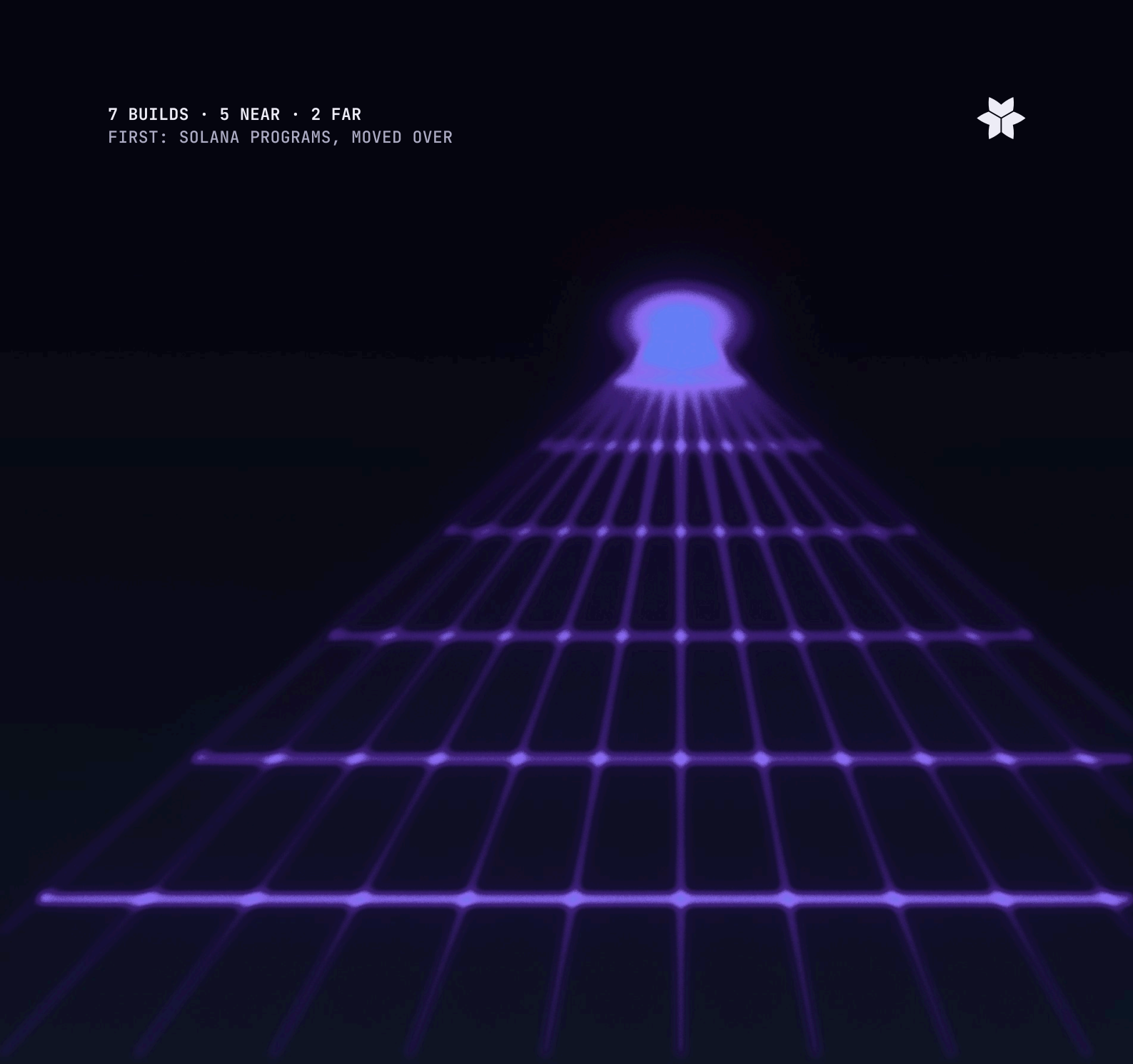
## 14.5 Fees on Acrux Net

Transaction fees on Acrux Net are paid in ACRUX to the proposers that carry a transaction; the rules and the fee-payer reserve are in Chapter 5.

Table 14.3 The three fees a transaction pays on Acrux Net, as `acrux-node` sets them.

| Fee | Charged | Goes to | On Acrux Net |
|---|---|---|---|
| Base fee | once, when selected | 50% burned, 50% to the carrying proposers | 5,000 lamports per signature |
| Priority fee | once, when selected | the carrying proposers | set by the sender, per CU |
| Inclusion fee | per included copy or fragment | the proposer whose block carries it | 1,000 lamports |

The carrying proposers split the unburned base fee and the whole priority fee equally. A plain send to one proposer costs 6,000 lamports, and a 2-of-3 threshold send 8,000 (Table 12.6).



15

# The road

Solana's apps and assets on Acrux, every fee and stake in ACRUX, and the chain run by independent operators. Seven builds on the way there.

Acrux is the SVM chain apps go to for 50 ms slots, and the one validators go to so they can propose those slots. Seven builds take it from the chain as it runs today to a chain run by independent operators, with Solana's apps and assets on it and every fee and stake in ACRUX.

## 15.1 The goal and how it is checked

Solana programs and the apps built on them run on Acrux with only their RPC URL changed. Assets come in from Solana over the bridge. Every fee and every stake on the chain is ACRUX. The chain is run by independent operators in hubs around the world.

The goal is reached when three things hold, and each can be looked up:

1. **The chain runs without its founders.** The project's genesis validators hold less than 20% of Acrux's stake, the byzantine share Alpenglow tolerates (Chapter 2), so the chain runs without them.
2. **Solana apps move by a URL.** An app that runs on Solana runs on Acrux with nothing changed but its RPC URL.
3. **ACRUX is the chain's money.** Fees and stake on Acrux are paid and held in ACRUX.

The road starts from everything this paper describes: the chain, the node, the Fault Lab, the Hub Registry on Acrux and on Solana, the SDK and the site. Each build stands on the one before it. Programs come first because the chain already runs SVM bytecode and every later build stands on Solana's tooling working against Acrux; the replay gives apps the number they ask for before they move; the bridge is how ACRUX and every other asset reach the chain; ACRUX becomes gas and stake once it can arrive; operators stake it once it is the stake; and the two far builds widen how a slot survives faults and how many regions propose it.

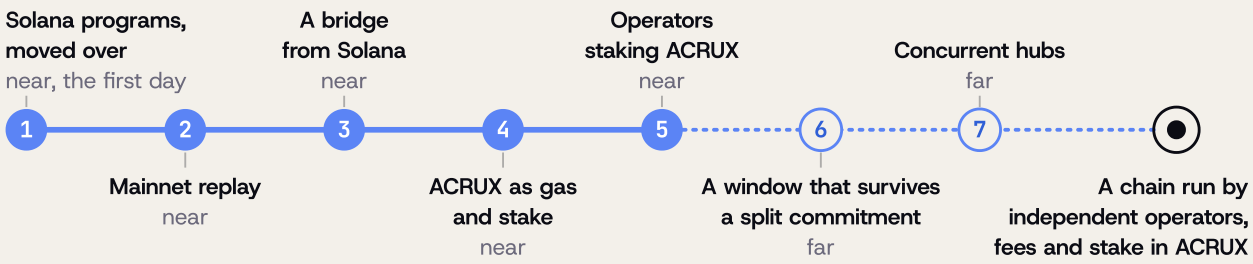



**Figure 15.1** The road. Builds 1 to 5 (solid) are near and stand on the chain as it runs; builds 6 and 7 (dotted) are far and carry the paper's Section 4 extensions into the protocol.

Table 15.1 The seven builds, what each opens, and how anyone checks it.

| # | Build | Reach | Opens | Checked by |
|---|---|---|---|---|
| 1 | Solana programs, moved over | near, the first day | apps that move without a rewrite | `solana --url` against Acrux; an Anchor suite passing |
| 2 | Mainnet replay | near | the number apps ask for before they move | `/replay` and `acrux replay` |
| 3 | A bridge from Solana | near | Solana's assets on Acrux, ACRUX first | a deposit on Solana and its credit on Acrux, and back |
| 4 | ACRUX as gas and stake | near | stake that operators outside the project hold | native supply equal to the ACRUX vault on Solana |
| 5 | Operators staking ACRUX | near | hubs formed bottom-up; the genesis share falling | `/net` marks external proposers; `/hubs` the genesis share |
| 6 | A window that survives a split commitment | far | the branch that becomes the protocol | the split-commitments scenario on each branch |
| 7 | Concurrent hubs | far | every region proposing in every slot | several hubs' blocks merged in one slot |

15.2

## Build 1: Solana programs, moved over

**What.** Acrux speaks Solana's JSON-RPC, so the `solana` CLI, Anchor, `@solana/web3.js` and `@solana/kit` work against it unchanged. A program is deployed with `solana program deploy`, and widely used open-source Solana programs run on Acrux at their mainnet addresses. "Solana at 50 ms" holds only if the SVM is Solana's own, programs and tools included.

**What a person gets.** A Solana developer takes a program and the client they already have, changes the RPC URL, and gets 50 ms slots without rewriting anything. Execution already runs on the SVM, and genesis already loads programs by address (`acrux net up --program ADDRESS=PATH`); what the build adds is the RPC surface the tools speak and the programs they expect to find.

**Checked by** `solana --url <acrux rpc>` commands run against the chain, an Anchor project's own test suite passing against Acrux, and a docs page listing every program on Acrux at its mainnet address with the slot it arrived in.

**Steps,** cut against the gateway (`gateway.rs`), `submission.rs`, `genesis.rs` and execution through LiteSVM:

1. **The Solana-shaped RPC.** A path on the gateway where the methods Solana tools call answer in Solana's shapes: `getVersion`, `getGenesisHash`, `getAccountInfo`, `getMultipleAccounts`, `getBalance`, `getLatestBlockhash`, `getSlot`, `getEpochInfo`, `getMinimumBalanceForRentExemption`, `getFeeForMessage`, `simulateTransaction`, `sendTransaction`, `getSignatureStatuses` and

`getTransaction`, beside the Acrux methods that share several of these names. `sendTransaction` goes through `submission.rs` as a plain t = 1 send to the target slot's proposers.

2. **Subscriptions.** `signatureSubscribe`, `accountSubscribe` and `slotSubscribe` in Solana's shapes on the events socket, so `sendAndConfirmTransaction` and wallet adapters confirm as they do on Solana.
3. `solana program deploy`. The upgradeable loader's buffer writes and deploy, executed through LiteSVM, then the program invoked.
4. **Anchor unchanged.** `anchor deploy` and `anchor test` with only the provider URL set to Acrux.
5. **Confirmation, measured.** The same transaction confirmed on Acrux and on Solana devnet from the same client, many times over, the two medians side by side in `web/lib/facts.ts`.
6. **Mainnet programs at their mainnet addresses.** Open-source programs whose licence allows it (Token-2022, Associated Token Account, Memo, and an AMM with its own SDK) dumped with `solana program dump` and placed at their mainnet addresses at an epoch boundary, each as its own step.
7. **The list.** A docs page on moving a program, with the programs on Acrux, their addresses and arrival slots.

---

15.3

## Build 2: mainnet replay

**What.** `acrux replay` and a new place, `/replay`: recorded Solana mainnet slots run through Acrux's merge (SortTxs) and selection (Algorithm 4, $m = 4$ tracks), beside the order their leader actually gave them, with every sandwich in those slots found and repriced as it would land on Acrux. No one gets the last look on Acrux; the replay shows what the last look takes from Solana's own traffic, and what is left of it on Acrux.

**What a person gets.** An app weighing a move, or a trader, opens any replayed mainnet slot and reads one number: what a sandwiched user would have kept on Acrux. It needs only the selection code the node already runs (Chapter 5) and recorded blocks, and it gives the bridge its reason.

**How the repricing works.** A sandwich is three transactions by one signer around one victim, on one pool in one slot: a front leg that buys what the victim buys, the victim, and a back leg that sells it.

— **The victim sent 2-of-3,** `/send`'s default. No proposer can read the victim in the slot it lands in, so the front leg has nothing to be built from and is dropped; the back leg survives as a back-run (paper Section 3.2). The victim's output is recomputed from the pool vaults' balances recorded in the block before the front leg (`preTokenBalances`), so no historical account state is needed.

— **A plain t = 1 victim**, a second column. The proposer that received it can still front-run it inside its own block, but only by outbidding the victim's priority fee per compute unit, and the replay subtracts that cost from the sandwich's take.

**Checked by** `/replay`, which lists the replayed range with its totals, each row linking the mainnet transaction on an explorer and its replayed slot; `acrux replay --block <file>` prints the same result in a terminal.

**Steps.** One slot and two orders (a `Replay` arm on the CLI and `acrux-node/src/replay.rs`, turning each recorded transaction into a `selection::Candidate` and running `sort_transactions` and `select_with_fees`); the sandwich finder, matching legs by signer, pool vault accounts and direction; repricing for constant-product pools, with `acrux replay --json` writing one record per slot; the read model (`ReplaySlot`, `ReplaySandwich`, `GET /v1/replay`, `GET /v1/replay/:slot`) and the two pages, the merged side drawn as `/net/slot/[slot]` draws it; a first contiguous range of recent mainnet slots with a `REPLAY` entry in `facts.ts`; concentrated-liquidity pools priced (Orca Whirlpools, Raydium CLMM, Meteora DLMM) by walking ticks or bins; and the range grown to a full Solana epoch.

---

## 15.4 Build 3: a bridge from Solana

**What.** SPL tokens move between Solana and Acrux. On Solana, an `acrux_bridge` program locks deposits in a vault per mint. On Acrux, the deposit is credited as the same token, at a mint derived from its Solana mint. Going back, a withdrawal burned on Acrux is proven to the Solana program and paid out of the vault. ACRUX is the first mint across. A faster Solana is used for what Solana's users already hold.

**What a person gets.** A holder moves tokens from a Solana wallet onto Acrux, trades and settles them in 50 ms slots, and moves them back.

**Mechanism.**

— **Solana to Acrux.** Every Acrux validator reads finalized Solana deposits through a Solana RPC, one configuration point per node. A proposer carries a deposit's credit in its block, and every validator checks the credit against the finalized deposit as one more validity condition beside Algorithm 2's (i) to (vii) (Chapter 4), so a credit with no deposit makes the block invalid.

— **Acrux to Solana.** Solana holds a light client of Acrux: each epoch's validator set and stakes, handed over by a certificate of the epoch before. A withdrawal proves its burn with a finalized cblock certificate, the validators' Ed25519 signatures checked through Solana's Ed25519 precompile, and a proof against an outbox root that the cblock's state hash commits to.

**Checked by** a deposit transaction on Solana mainnet and its credit on Acrux, a burn on Acrux and its payout on Solana, and the vault's balance per mint, readable on Solana.

**Steps.** `acrux_bridge` on the local validator with `deposit`, the vault per mint and the deposit record; the credit on Acrux, with the deposit reader in the node, the validity condition and the derived mint, every validator's state hash agreeing; the outbox root in the state hash and the light client on Solana with its epoch handover; withdrawal, a token's round trip from Solana to Acrux and back; `/bridge`, to deposit with a Solana wallet and withdraw with the Acrux key; and the program on Solana mainnet with the first mainnet deposit of ACRUX credited on Acrux.

## 15.5 Build 4: ACRUX as gas and stake

**What.** At one epoch boundary, the chain's native unit becomes ACRUX that came across the bridge. From then on:

- **Gas.** The base fee, priority fee, inclusion fee and the fee-payer reserve $\rho$ are paid in ACRUX (Chapter 14). Native balance on Acrux exists only as ACRUX credited against the Solana vault, and the genesis faucet's balance is retired at that boundary.
- **Stake.** Stake is ACRUX staked to validator identities through the stake program Solana uses, and Algorithm 3 reads proposer stake from it (Chapter 6).

Acrux is its own chain, and its fees and its security are held in its own token: an ACRUX holder pays for transactions in it and stakes it to the validators who propose Acrux's slots.

**Checked by** a transaction's fee shown in ACRUX on `/tx/[id]`, the epoch of the cut-over and the stake at it on `/hubs`, and native supply on Acrux matching the ACRUX vault's balance on Solana.

**Steps.** The cut-over epoch in the genesis configuration, with the deposit credit for ACRUX minting native units; two values there are configuration points, the scale from the ACRUX mint's decimals onto the 9-decimal native unit and the genesis validators' stake at the cut-over. Then fees and the reserve in bridged ACRUX through `acrux-execution`; stake accounts on Acrux feeding `acrux_schedule::cap_stakes` and `build_schedule`; and the cut-over on the chain, the first epoch run on ACRUX stake.

## 15.6 Build 5: operators staking ACRUX

**What.** A validator outside the project runs `acrux-node` with its own identity, stakes ACRUX to it, joins a hub through the Hub Registry on Acrux, and proposes. Its blocks are sealed, committed, attested and merged like the genesis nodes' blocks. A Solana validator that has joined a hub on the Solana

registry is admitted with the identity it already holds (Chapter 11). A slot authored only by the project's own nodes has a leader by another name; the chain is what this paper describes only when its $k$ proposers are independent.

**What a person gets.** An operator proposes Acrux's slots from its own data centre and earns their fees. It learns, by measurement, whether its machine keeps within $2\delta_{\text{local}}$ of its hub, and everyone else gets $\delta_{\text{local}}$ measured between independent operators, where today it is applied on the wire (Chapter 9). Following the chain already works with `acrux node follow` (Chapter 13); proposing from an outside machine is what the build adds.

**Checked by** `/net`, which marks each proposer as genesis or external; `/hubs/[code]`, which shows each member pair's measured hash-exchange time; and `/hubs`, which shows the genesis validators' share of stake each epoch as it falls toward the goal.

**Steps.**

1. `acrux node propose --hub FRA --identity <keypair>`: a command beside `node follow` that runs one validator on its own machine, with `acrux-transport` opening QUIC to its hub's co-proposers at their public addresses and no hub delay applied to that link.
2. Admission: an identity with ACRUX stake and an effective membership in the registry on Acrux becomes a member of its hub at the next epoch boundary; memberships on the Solana registry carry over for identities that sign on Acrux.
3. The first external block: the node drawn by `build_schedule`, its block committed inside $2\delta_{\text{local}}$ and merged.
4. Measured $\delta_{\text{local}}$: each node exports its hash-exchange time to each co-proposer (`exchange.rs`), aggregated per hub and per member pair, with the commit spread in `facts.ts` split into applied and real links.
5. A hub formed by its members: registered and joined only by external operators, active once it meets $n_{\min} = 4$ and $\rho_{\min} = 1\%$, with windows going to it.
6. The genesis share each epoch, on `/hubs`.

---

## 15.7 Build 6: a window that survives a split commitment

**What.** Two research branches of the node, each answering the one fault that costs Acrux a window: a proposer that sends its commitment to only some of its co-proposers, which makes the vectors disagree and skips the rest of the window for everyone (Chapter 10).

— **Local agreement** (paper Section 4.1): a one-shot agreement among the window's proposers on the set of commitments.

— **Lanes** (Section 4.2): one Alpenglow instance per proposer, which also keeps a window alive at $k = 2$.

On a chain whose product is the 50 ms slot, one byzantine proposer does not get to stop a window: under that attack, users' transactions keep landing in the next slots. The build weighs each branch's cost ($k$ consensus instances, asynchronous execution, more votes) once independent operators run the chain, and settles which branch becomes the protocol. Lanes also free the decision from co-location, which build 7 stands on.

**Checked by** the Fault Lab's split-commitments scenario (`acrux lab run`) on each branch: slots skipped, votes per cblock and finalization time, side by side on `/lab/[run]`.

## 15.8 Build 7: concurrent hubs

**What.** Several hubs propose in the same slot, each running its own hash exchange and attestations, and their blocks are merged into one order (paper Section 4.3). With one hub per window, one region holds each slot; with concurrent hubs every region proposes in every slot, and censoring a transaction takes proposers in several jurisdictions.

**What a person gets.** A user's transaction reaches a proposer in its own region in every slot, not only when the schedule passes through it. The build stands on build 6's Lanes and on hubs that really are far apart (build 5), and it settles the two questions the paper sets for it: hiding at global distances, so that one hub cannot see the others' blocks before committing its own, and how a hub far from most of the stake keeps its blocks on time.

## 15.9 The first day

Build 1's steps are the first day. Each lands on its own, and each is something people can run themselves:

1. `solana balance` and `solana transfer` against Acrux.
2. A web3.js send-and-confirm.
3. A program deployed with `solana program deploy`.
4. An Anchor test suite passing, unchanged.
5. Confirmation time on Acrux beside Solana's.
6. Token-2022, Associated Token Account, Memo and an AMM at their mainnet addresses, one by one.
7. The list of programs on Acrux.

Steps 1 to 4 touch only the gateway, `submission.rs` and the loader path through LiteSVM; step 5 is a measurement, and steps 6 and 7 are deploys and a page. Nothing in them waits on anyone outside the project.

> The chain runs Solana's programs at 50 ms today; the road brings Solana's apps, its assets and independent operators onto it.



# Appendices

Parameters, every interface the node and the program expose, the wire formats, and each line of the paper beside the code that runs it.

Appendix A

# Parameters

The parameters Acrux Net runs with. The first table is the paper's Table 1 as acrux-node sets it; the second holds the values the paper leaves open and the node decides.

## A.1 Protocol parameters

Every value below is the default of `acrux-node`, the configuration of Acrux Net, and the value `backend/spec/map.json` records beside the code. The derived rows follow from the others: a window is $w$ slots of $\delta_{\text{block}}$, the commit exchange window is $2\delta_{\text{local}}$, and the base timeout is $3\delta_{\text{global}}$.

Table A.1 New Constellation's parameters as Acrux Net runs them [1].

| Symbol | Parameter | Value | Source |
|---|---|---|---|
| $k$ | Concurrent proposers | 3 | Paper, Table 1 |
| $\kappa$ | Slots per consensus slot | 4 | Paper, Table 1 |
| $w$ | Slots per proposer window | 16 | Paper, Table 1 |
| $\delta_{\text{local}}$ | Maximum local network delay | 10 ms | Paper, Table 1 |
| $\delta_{\text{block}}$ | Length of a slot | 50 ms | Paper, Table 1 |
| $\delta_{\text{global}}$ | Maximum network delay | 250 ms | Paper, Table 1 |
| $w \cdot \delta_{\text{block}}$ | Length of a window | 800 ms | Derived |
| $2\delta_{\text{local}}$ | Commit exchange window | 20 ms | Derived |
| $\Delta_{\text{timeout}}$ | Alpenglow base timeout | 750 ms | Acrux: $3\delta_{\text{global}}$ |
| $m$ | Execution tracks | 4 | Paper, Section 3.3 |
| $\rho$ | Fee-payer reserve | 0.001 ACRUX | Paper, Section 3.4 |
| $\alpha$ | Share of windows drawn by hub stake | 1/2 | Paper, Algorithm 3 |
| | Notarization threshold | 60% | Alpenglow |
| | Fast finalization threshold | 80% | Alpenglow |
| | Shreds per slice | 64 | Rotor |
| | Shreds to rebuild a slice | 32 | Rotor |

## A.2 Values the paper leaves open

These are set by `acrux-node` and can be changed through its environment (`.env.example`) or configuration. Each one is told in the chapter that uses it.

Table A.2 Values acrux-node sets where the paper leaves a choice.

| Symbol | Parameter | Value | Set by |
|---|---|---|---|
| $b_{cu}$ | Compute units per track per slot | 2,000,000 CU | `ACRUX_BCU` |
| $m \cdot b_{cu}$ | Compute units per slot | 8,000,000 CU | Derived |
| | Fragment expiry after the first accepted fragment | 8 slots (400 ms) | `ACRUX_FRAGMENT_EXPIRY` |
| $n_{min}$ | Members for an active hub | 4 | Schedule |
| $\rho_{min}$ | Stake for an active hub, share of total | 1% | Schedule |
| | Base fee per signature, half burned | 5,000 lamports | `ACRUX_BASE_FEE` |
| | Inclusion fee per included copy or fragment | 1,000 lamports | `ACRUX_INCLUSION_FEE` |
| $\rho$ | Fee-payer reserve in lamports | 1,000,000 | `ACRUX_RESERVE` |
| | Blockhash validity | 1,200 slots (one minute) | Gateway |
| | Blockhash returned by `getLatestBlockhash` | 8 slots back | Gateway |
| | Slots kept in the run store | 24,000 | `ACRUX_RETAIN_SLOTS` |

## A.3 Choices of construction

Where the paper names a function and not its construction, Acrux fixes one. Each is described where it is used: the cipher and the commitment in Chapter 8, the codes in Appendix D, the schedule's draw in Chapter 6.

**Table A.3** Constructions acrux-node fixes.

| Function | Construction |
|---|---|
| Hash | SHA-256 |
| Signatures | Ed25519 |
| Block cipher | ChaCha20-Poly1305, one fresh key per block, H(K) bound in the commitment |
| Shreds and fragments | Reed-Solomon erasure coding, 64 shreds per slice of which any 32 rebuild it, Merkle proofs per shred |
| Key shares | Shamir secret sharing over GF(256) |
| Transport | QUIC, one endpoint per validator, each link delayed by its hub geography |
| Schedule randomness | An epoch-seeded ChaCha20 draw |
| Regional runs | Drawn windows kept contiguous by region, after SIMD-0675 [12] |
| Execution | The Solana Virtual Machine through LiteSVM |

Appendix B

# Interfaces: JSON-RPC, events, HTTP API, CLI

Every interface Acrux serves, as reference: the gateway's JSON-RPC methods and the types they carry, the event stream, the gateway's HTTP routes, the HTTP API, and every command and flag of the `acrux` CLI. Every shape is a schema of `@acrux/contract`; the API serves the same reference as OpenAPI 3.1 at `GET /openapi.json`.

## B.1 JSON-RPC

The Acrux Net gateway answers JSON-RPC 2.0 over HTTP POST at `/` or `/rpc`, locally on `http://127.0.0.1:4930`. A request is `{ "jsonrpc": "2.0", "id", "method", "params" }`; `id` is a string or a non-negative integer and comes back unchanged; `params` is always an object, `{}` for a method without params, and unknown keys are refused. A success is `{ "jsonrpc", "id", "result" }`; a failure is `{ "jsonrpc", "id", "error": { "code", "message", "data" } }`, where `data` is the `ApiError` envelope of Table B.10. Numbers that can exceed $2^{53}$ (lamports, stake, priority fees) are decimal strings; times are integer microseconds from the anchor their field names; proposer and track indices start at 1, slice, fragment and merged-order indices at 0; a `null` in an attestation vector `pi` is the paper's $\perp$.

Table B.1 JSON-RPC error codes and the `ErrorCode` each carries in `data.error.code`.

| `error.code` | When | `data.error.code` |
|---|---|---|
| -32700 | the body is not JSON | `INVALID_INPUT` |
| -32600 | not a JSON-RPC 2.0 request with an `id` | `INVALID_INPUT` |
| -32601 | unknown method | `NOT_FOUND` |
| -32602 | `params` fail validation; `data.error.fieldErrors` names each field | `INVALID_INPUT` |
| -32000 | the method ran and refused; `error.message` repeats the code | any `ErrorCode` |

Table B.2 Every JSON-RPC method of the gateway: its params, its result, and what it returns.

| Method | Params | Result | Returns |
|---|---|---|---|
| `getSnapshot` | `{}` | `NetSnapshot` | the whole current view: stream position, status, current window, last 64 slots, windows ahead, counters, parameters |
| `getSlot` | `slot` | `Slot` | one slot, complete; `NOT_FOUND` for a slot not held |
| `getRecentSlots` | `limit` 1 to 64, default 64 | `Slot[]` | the most recent slots, oldest first, including those in progress |

| Method | Params | Result | Returns |
|---|---|---|---|
| `getCurrentWindow` | `{}` | `Window` | the window producing now |
| `getSchedule` | `limit` 1 to 64, default 8 | `Window[]` | the windows after the current one, in order |
| `getCounters` | `{}` | `Counters` | net-wide counters |
| `getFacts` | `{}` | `Facts` | the last ten minutes measured on the node clock; `NET_UNAVAILABLE` until enough slots are measured |
| `getTransaction` | `id` | `Transaction` | a transaction's trail; state `unknown` for an ID never seen |
| `getBalance` | `address` | `Balance` | lamports, the reserve held, and what is spendable above it |
| `getExecutionSnapshot` | `{}` | `{ consensusSlot, executor, threshold }` | the executed state of the latest finalized cblock and the fragment cache, where `acrux node follow` starts; `NET_UNAVAILABLE` before the first finalized cblock |
| `getSlotBlocks` | `{ slot }` | the agreed blocks, each `{ proposer, slices }` | a slot's encrypted blocks as sealed, for re-execution; `NOT_FOUND` past the last 64 slots |
| `getDemoPool` | `{}` | `DemoPool` | the demo constant-product pool: mints, reserves, fee |
| `getQuote` | `inputMint`, `amountIn`, `slippageBps` 0 to 10,000 | `Quote` | a swap quote at the current reserves; `INVALID_INPUT` for a mint not in the pool |
| `getLatestBlockhash` | `{}` | `blockhash`, `slot`, `startedSlot` | the blockhash from 8 slots before the latest executed slot; any of the last 1,200 slots is valid |
| `submitTransaction` | `PlainSubmission` | `Submission` | a plain transaction ($t = 1$) for its target slot |
| `submitFragment` | `FragmentSubmission` | `Submission` | one fragment of a $t$-of-$n$ transaction, for one receiver |
| `getLabReplay` | `runId` | `LabReplay` | a Fault Lab run's recorded events, served by the API; `NOT_FOUND` on the live node |

`submitTransaction` and `submitFragment` refuse with `INVALID_INPUT` (a receiver named twice; `t > n` or `index ≥ n`; a target slot more than 64 slots ahead), `TRANSACTION_EXPIRED` (the target slot has started), `CHAIN_REFUSED` (the node refused it; `message` says why) and `IDEMPOTENCY_CONFLICT` (the ID submitted before with a different header, body, `n` or `t`, or this index with a different fragment). Submitting the same transaction again with the same header and body returns the same result. The public host limits reads and submits per client address (Table 13.6).

**Table B.3** The types the gateway's methods and events carry, with their fields in order, then the values of their enumerations. Every field is present; `null` marks a value not yet known.

| Type | Fields |
|---|---|
| NetSnapshot | streamId, cursor, status, currentWindow, recentSlots, upcomingWindows, counters, parameters |
| Parameters | k, kappa, w, deltaLocalUs, deltaBlockUs, deltaGlobalUs, timeoutUs, tracks, reserveLamports, trackCapacityCu, fragmentExpirySlots, alpha, minimumHubMembers, minimumHubStake |
| Counters | slots, finalizedCblocks, skips, averageCommitSpreadUs |
| Window | number, hubCode, firstSlot, lastSlot, consensusSlots, proposers |
| Proposer | index, identity, stake, status |
| Slot | slot, wallTime, window, consensusSlot, status, blocks, attestations, agreedSet, skip, mergedOrder, schedule, stateHash, xblockHash, cblock, executedUs, attestationDeadlineUs, consensusTimeoutUs |
| ProposerBlock | proposer, identity, status, bytes, slices, commitment, attestation, parent |
| Slice | index, shredCount, rebuildShreds, bytes, sentUs, arrivedUs, merkleRoot |
| Commitment | slot, proposer, hash, keyHash, signature, sentUs, arrivals |
| CommitmentArrival | receiver, startUs, arrivedUs, relativeToStartUs, deadlineUs, onTime |
| Attestation | slot, proposer, pi, key, signature, sentUs, validatorArrivals |
| Skip | reason, detail, disagreeingVectors, affectedSlots, atUs |
| MergedTransaction | id, feePayer, priorityFeePerCu, computeLimit, writableAccounts, targetSlot, kind, sourceProposers, n, t, fragmentIndices, position, selected, exclusion |
| ExecutionSchedule | algorithm, tracks, capacityCu, placements, gaps |
| Placement | track, startCu, lengthCu, txId |
| Cblock | consensusSlot, slots, parent, hash, stateHash, votes, certificates, status, finalizedUs |
| Vote | consensusSlot, validator, kind, stakeShare, hash, atUs |
| Certificate | consensusSlot, kind, stakeShare, threshold, hash, signers, atUs, linkedConsensusSlot |
| Transaction | header, state, kind, n, t, receivers, fragments, includedCount, readableSlot, executedSlot, mergedPosition, result, logs, failure, expiresAfterSlot, finalization |
| TxHeader | id, feePayer, priorityFeePerCu, computeLimit, writableAccounts, targetSlot |
| FragmentInclusion | index, receiver, slot, proposer, includedUs |
| Balance | address, lamports, reserveLamports, spendableLamports, slot |
| DemoPool | address, mints, reserves, feeBps, slot |
| DemoMint | address, symbol, name, decimals |
| Quote | inputMint, outputMint, amountIn, amountOut, minimumOut, feeAmount, slot |

| Type | Fields |
|---|---|
| PlainSubmission | header, transaction, receivers |
| FragmentSubmission | header, n, t, index, receiver, fragment, keyShare, headerSignature |
| Submission | id, accepted, targetSlot |
| SkipReason | disagreeing_vectors, missing_attestation, equivocation, majority_lost, window_skipped |
| Slot.status | pending, executed, finalized, skipped |
| Transaction.state | unknown, sent, partly_included, executed, finalized, failed, expired |
| Proposer.status | online, late, crashed |
| ProposerBlock.status | streaming, sealed, accepted, dropped, missing |
| MergedTransaction.exclusion | capacity, priority_fee, incomplete, null |
| Vote.kind | notar, notar_fallback, skip, skip_fallback, finalize |
| Certificate.kind | notarization, notar_fallback, fast_finalization, slow_finalization, skip |
| Cblock.status | pending, notarized, finalized, skipped |

## B.2 Event stream

The gateway streams every protocol event over a WebSocket, locally on `ws://127.0.0.1:4931`. On connecting it sends a `stream.snapshot` with the complete current view. Every `subscribe` is answered with a `stream.resync` carrying a fresh snapshot, or with the retained events after the cursor it names; on any `stream.resync` a client replaces its state with the snapshot. `stream.*` events reach every subscriber whatever its filter.

Table B.4 Commands a client sends on the socket, and the filters a subscribe takes.

| Command or filter | Effect |
|---|---|
| `{ "version": 1, "type": "subscribe", "filter": … }` | start delivery for `filter`; replaces any earlier filter |
| `… "afterSequence": n, "streamId": "…"` | resume: the retained events after sequence `n` of that stream |
| `{ "version": 1, "type": "unsubscribe" }` | stop delivery |
| `{ "kind": "all" }` | every event |
| `{ "kind": "slots", "from": a, "to": b }` | every event whose `slot` is from `a` to `b` inclusive |
| `{ "kind": "transaction", "id": "…" }` | `transaction.updated` for that ID, and the `slot.started`, `block.decrypted`, `slot.merged`, `slot.executed` and `lab.evidence` events that contain it |

**Table B.5** The envelope every event carries, and the anchors its time is measured from.

| Field | Type | Meaning |
|---|---|---|
| `version` | `1` | envelope version |
| `node` | string, optional | the identity of the node that emitted the event, where one node did |
| `sequence` | integer | position in the stream, rising by one per event within a `streamId` |
| `streamId` | string | the stream instance; a new one means earlier sequences no longer apply |
| `type` | string | the event type |
| `slot` | integer or null | the slot the event belongs to; `null` for stream and window control |
| `anchor` | `slot`, `window`, `consensus`, `stream` | `atUs` counts from the slot's start (slices, commitments, attestations, merge, execution), the window's ParentReady (window events, deadlines, timeouts), the cblock's first slot (votes, certificates), or the stream's start |
| `atUs` | integer | microseconds after the anchor; each slot carries one UTC `wallTime` |
| `data` | object | the payload, by type |

**Table B.6** Every event type, in the order the events occur within a slot.

| `type` | `data` | Meaning |
|---|---|---|
| `stream.snapshot` | `NetSnapshot` | first message on connect: the complete current view |
| `stream.resync` | `reason`: connected, cursor_expired, restarted, filter_changed; `snapshot` | replace local state with `snapshot` |
| `stream.heartbeat` | `latestSlot`; `status`: live, stalled, unavailable | the latest slot and the net's status |
| `stream.error` | `ApiError` | the stream failed |
| `window.started` | `Window` | a window started: hub, slots, consensus slots, proposers |
| `slot.started` | `Slot` | a slot started, still pending |
| `block.slice` | `proposer`; `slice`: Slice | a proposer sent a slice: 64 shreds, any 32 rebuild it |
| `commitment.sent` | `Commitment` | a proposer sent its signed commitment with its last slice |
| `commitment.arrived` | `proposer`; `hash`; `arrival`: CommitmentArrival | a commitment reached a co-proposer, on time or late |
| `attestation.sent` | `Attestation` | a proposer attested and released its key |
| `attestation.arrived` | `attestation`; `validator`; `arrivedUs` | an attestation reached a validator |
| `slot.agreed` | `agreedSet` | the slot is agreed on the set *P* |
| `slot.skipped` | `Skip` | the slot will not execute: the reason and the evidence |
| `block.decrypted` | `proposer`; `transactionIds` | a block of *P* was decrypted |
| `slot.merged` | `transactions`: MergedTransaction[] | the slot's transactions in their one order |

| type | data | Meaning |
|---|---|---|
| slot.scheduled | ExecutionSchedule | Algorithm 4's placement on the four tracks, with the gaps |
| slot.executed | Slot | the slot executed, with its state hash |
| cblock.assembled | Cblock | four xblocks, the parent and the state hash form a cblock |
| cblock.vote | Vote | a validator voted on a cblock |
| cblock.certificate | Certificate | votes reached a threshold |
| cblock.finalized | Cblock | a cblock is final |
| transaction.updated | Transaction | a transaction's trail changed |
| net.counters | Counters | the net's counters |
| lab.updated | LabRun | a Fault Lab run changed status |
| lab.evidence | Evidence | a Fault Lab run recorded evidence |

Within one slot of a window with three correct proposers a subscriber sees slot.started; the block.slice events of the three blocks, interleaved as they stream; commitment.sent from each proposer with its last slice, then commitment.arrived at each co-proposer; attestation.sent once each exchange closes, then attestation.arrived at validators; slot.agreed or slot.skipped; block.decrypted per block of the agreed set; slot.merged, slot.scheduled, slot.executed; and after the fourth slot of a cblock cblock.assembled, the cblock.vote events, cblock.certificate and cblock.finalized. Slots are pipelined, so events of neighbouring slots interleave; order by slot and sequence.

The gateway keeps the most recent 24,000 events and 5,000 transaction records. A resume with a retained cursor receives the events after it; otherwise a stream.resync arrives with reason restarted when the streamId is no longer the gateway's, filter_changed otherwise, and a client that falls behind the live stream gets reason cursor_expired. A Fault Lab run's events use the same envelope and types.

## B.3 Gateway HTTP routes

Table B.7 The node's HTTP routes. The metrics port is private on the public host and never proxied.

| Port | Route | Serves |
|---|---|---|
| 4930 | `POST /`, `POST /rpc` | JSON-RPC |
| 4930 | `GET /health` | `status` (`ready` once a window is producing, `waiting` before), `contractVersion`, `missingConfiguration` |
| 4931 | `GET /`, `GET /events` | the WebSocket upgrade of the event stream |
| 4989 | `GET /metrics` | Prometheus text 0.0.4: `acrux_slots_total`, `acrux_finalized_cblocks_total`, `acrux_skips_total`, `acrux_commit_spread_microseconds` |
| 4989 | `GET /health` | as on the gateway port |
| 4989 | `POST /internal/faucet` | the faucet's grant route, called only by the API |

## B.4 HTTP API

The API serves what is not the live net itself: the Hub Registry's hubs and members as indexed from Solana, the Algorithm 3 schedule preview, transaction lookup, the faucet, Fault Lab runs and their replays, the figures measured on the net, and the map from the paper to the code. Locally it is `http://127.0.0.1:4920`, typed by `createClient` of `@acrux/contract`. Every response is JSON and every failure is the `ApiError` envelope with the HTTP status its code carries. Lists are paged: `cursor` and `limit` (1 to 100, default 24), following `page.nextCursor` until `page.hasMore` is false.

Table B.8 Every route of the HTTP API, with its `createClient` call beneath it, and what it returns.

| Route and client call | Returns | Notes |
|---|---|---|
| `GET /health`<br>`health()` | `Health` | status and the configuration still missing |
| `GET /v1/hubs`<br>`hubs(query)` | `Hubs` | every hub, active, pending and inactive; `cursor`, `limit` |
| `GET /v1/hubs/{code}`<br>`hub(code)` | `Hub` | one hub; `NOT_FOUND` if no hub holds the code |
| `GET /v1/registry/config`<br>`registryConfig()` | `RegistryConfig` | program ID, cluster, deposit, fee, cooldown, inactivity, code rules, minimums, current epoch |
| `GET /v1/registry/members/{identity}`<br>`member(identity)` | `Member` | a membership; `NOT_FOUND` if in no hub |
| `GET /v1/schedule`<br>`schedule(epoch?)` | `SchedulePreview` | Algorithm 3 for `epoch`, the next epoch by default |

| Route and client call | Returns | Notes |
|---|---|---|
| `GET /v1/transactions/{id}`<br>`transaction(id)` | `Transaction` | the trail across the node's store, as `getTransaction` |
| `GET /v1/faucet/{address}`<br>`faucetStatus(address)` | `FaucetStatus` | policy, grants left for the key and the IP, reset time |
| `POST /v1/faucet`<br>`faucet({ address })` | `FaucetGrant` | test ACRUX with 100 AXA and 100 AXB; `FAUCET_LIMIT`, `CONFIGURATION_REQUIRED`, `VALIDATION_ERROR` |
| `GET /v1/lab/scenarios`<br>`labScenarios()` | `LabScenario[]` | the six scenarios with seed, parameters, lemma, expected outcome |
| `GET /v1/lab/runs`<br>`labRuns(query)` | `LabRuns` | recorded runs, newest first; `cursor`, `limit` |
| `POST /v1/lab/runs`<br>`startLabRun({ scenario, seed? })` | `LabRun` | starts a run; `201` with status `queued` |
| `GET /v1/lab/runs/{id}`<br>`labRun(id)` | `LabRun` | status, outcome, evidence |
| `GET /v1/lab/runs/{id}/events`<br>`labReplay(id)` | `LabReplay` | every recorded event, for replay |
| `GET /v1/lab/runs/{id}/stream`<br>`labStreamUrl(id)` | server-sent events | recorded then live events; `event` the type, `id` the sequence; resumes from `Last-Event-ID` |
| `GET /v1/facts`<br>`facts()` | `Facts` | the figures measured on Acrux Net, p50 and p99, with handover per hub link |
| `GET /v1/spec`<br>`spec()` | `SpecMap` | every algorithm line and lemma with its file, function and scenario; the decisions |
| `GET /openapi.json` | OpenAPI 3.1 | the API, with the gateway's JSON-RPC and event stream as `x-acrux-json-rpc` and `x-acrux-event-stream` |

**Table B.9** The types of the HTTP API, with their fields in order, then the values of their enumerations.

| Type | Fields |
|---|---|
| `Health` | `status, contractVersion, missingConfiguration` |
| `Hubs` | `items, page, asOf` |
| `Hub` | `code, name, region, ecef, latitude, longitude, registrant, depositLamports, members, stake, proposerStake, proposerStakeCap, status, needs, createdEpoch, lastActiveEpoch, releasableEpoch, releasable` |
| `Member` | `identity, hubCode, stake, proposerStake, capped, effectiveEpoch, joinedEpoch, pendingMove, pendingLeave, cooldownUntilEpoch` |
| `RegistryConfig` | `status, programId, cluster, depositLamports, registrationFeeBps, cooldownEpochs, inactivityEpochs, codeRules, minimumMembers, minimumStake, currentEpoch, missingConfiguration` |
| `SchedulePreview` | `epoch, seed, source, algorithm, alpha, totalWindows, shares, regionalRuns, windows` |
| `FaucetStatus` | `address, policy, remainingForKey, remainingForIp, resetsAt, eligible` |
| `FaucetPolicy` | `amountLamports, perKey, perIp, periodSeconds, missingConfiguration` |

| Type | Fields |
|---|---|
| FaucetRequest | address |
| FaucetGrant | id, transactionId, address, amountLamports, signature, balance, grantedAt |
| LabScenario | id, title, description, seed, parameters, lemma, expectedOutcome |
| LabRunRequest | scenario, seed |
| LabRuns | items, page, asOf |
| LabRun | id, scenario, seed, status, createdAt, startedAt, endedAt, outcome, evidence, failure |
| Evidence | kind, slot, proposer, txId, description, hashes, signatures, observedFragments, threshold |
| LabReplay | run, events |
| Facts | source, measuredAt, sampleSize, slotPacingUs, commitSpreadUs, attestationRelativeToBlockUs, cblockFinalizationUs, handoverUs |
| SpecMap | paper, parameters, entries, decisions |
| ScenarioId | last-look, censorship, crash, equivocation, split-commitments, private-front-run |
| Evidence.kind | late_block_dropped, transaction_included, two_proposers_continue, slashing_proof, skip_certificate, unreadable_fragment |
| Hub.status | active, pending, inactive |
| LabRun.status | queued, running, done, failed_to_start |

Table B.10 The `ApiError` envelope: every failed API response, and the `data` of every JSON-RPC error.

| Field | Type | Meaning |
|---|---|---|
| error.code | ErrorCode | the stable code to branch on |
| error.message | string | a sentence to show |
| error.retryable | boolean | whether the same request may succeed later |
| error.stage | string | the step that failed: `validation`, `auth`, `read`, `upload`, `build`, `wallet`, `co_sign`, `simulation`, `send`, `confirmation` |
| error.fieldErrors | map, optional | for invalid input: each field path and what is wrong with it |
| error.programError | object, optional | for a refused Solana transaction: `programId`, `name`, `number` |
| error.retryAfterSeconds | number, optional | when to retry, for limits |
| error.requestId | string, optional | the request's ID, for reports |

Table B.11 Every `ErrorCode` of every Acrux service, with the HTTP status it carries.

| Code | HTTP | Meaning |
|---|---|---|
| `VALIDATION_ERROR` | 400 | the request failed validation; see `fieldErrors` |
| `UNAUTHORIZED` | 401 | the request needs credentials it did not carry |
| `ORIGIN_NOT_ALLOWED` | 403 | a browser origin the service does not serve |
| `NOT_FOUND` | 404 | no such hub, member, slot, transaction, run or method |
| `PROGRAM_UNAVAILABLE` | 409 | the Hub Registry program is not deployed or not reachable |
| `CONFIGURATION_REQUIRED` | 409 | the service lacks a setting the call needs |
| `IDEMPOTENCY_CONFLICT` | 409 | the same ID was submitted before with different content |
| `TRANSACTION_EXPIRED` | 409 | the target slot has started, or the transaction is already settled |
| `PAYLOAD_TOO_LARGE` | 413 | the body exceeds the limit |
| `UNSUPPORTED_MEDIA_TYPE` | 415 | the body is not JSON |
| `SIMULATION_FAILED` | 422 | a Solana transaction failed simulation |
| `CHAIN_REFUSED` | 422 | a Solana transaction was refused by a program; see `programError` |
| `RATE_LIMITED` | 429 | too many requests; see `retryAfterSeconds` |
| `INTERNAL_ERROR` | 500 | the service failed |
| `UPSTREAM_UNAVAILABLE` | 502 | a service this one depends on did not answer |
| `FAUCET_LIMIT` | 429 | this key or IP has used its faucet grants for the period |
| `NET_STALLED` | 503 | Acrux Net has stopped producing slots |
| `NET_UNAVAILABLE` | 503 | the gateway could not be reached |
| `INVALID_INPUT` | 400 | the gateway refused the params or the submission |

## B.5 CLI

`acrux` is the node's binary; `acrux <command> --help` prints the same options. Building it is in Section 13.2.

Table B.12 Every command of `acrux`, its arguments and options with their defaults, and what it does and prints.

| Command | Options and defaults | Does and prints |
|---|---|---|
| `acrux net up` | the flags of Table 13.1: `--gateway-port` 4930, `--ws-port` 4931, `--base-port` 4950, `--metrics-port` 4989, `--traffic-rate` 50, `--run-seconds`, `--crash-window`, `--crash-proposer` 3, `--program ADDRESS=PATH`, `--genesis`, `--store target/runs/net` | runs a twelve-validator net in FRA, NYC and TYO with the gateway and metrics; on stop prints one JSON line: `report`, `finalizedCblocks`, `skips`, `slotPacingUs` |

| Command | Options and defaults | Does and prints |
|---|---|---|
| `acrux cluster up` | `--hubs FRA,NYC,TYO`, `--gateway-port`, `--ws-port`, `--base-port`, `--metrics-port`, `--store target/runs/cluster` | the same net, named by its hubs, with the genesis at `../chain/genesis/acrux-net.json` |
| `acrux keygen` | `--out acrux-key.json` | writes a Solana keypair file; prints its address |
| `acrux send` | `--keypair`, `--to`, `--lamports` required; `--n` 1, `--t` 1, `--priority` 1000 micro-lamports per CU, `--rpc http://127.0.0.1:4930` | a transfer with a 20,000 CU limit for the slot four after the latest started, plain or sealed; prints `sent <id> for slot <slot>`, then each state to `finalized`, `failed` or `expired` |
| `acrux node follow URL` | the gateway's JSON-RPC URL | starts from the latest finalized cblock's executed state, checks every finalized slot's attestation signatures and agreed set, decrypts, merges and re-executes it, and checks its state hash and finalization certificate; `--genesis` names the net's genesis for its programs (default `../chain/genesis/acrux-net.json`); prints one line per slot; stops on a failed check (Table 13.4) |
| `acrux hub register CODE` | `--latitude`, `--longitude` in degrees; the hub options | `register_hub`, the location as WGS84 ECEF at sea level |
| `acrux hub join CODE` | the hub options | `join_hub`, the identity as payer too |
| `acrux hub leave CODE` | the hub options | `leave_hub` |
| `acrux hub move FROM TO` | the hub options | `move_hub` |
| `acrux hub release CODE` | the hub options | `release_hub`, the registrant read from the hub account |
| `hub options` | `--identity acrux-key.json`, `--rpc http://127.0.0.1:4990`, `--program` the registry id | each hub command prints `<instruction> confirmed: <signature>`, waiting up to 30 s |
| `acrux schedule` | JSON on stdin: `{ epoch, seed, windows, totalStake, hubs }` | runs Algorithm 3 on the code path the node's own schedule runs; prints `{ hubStakes, schedule }` |
| `acrux lab run` | `--scenario`, `--seed`, `--out` required; `--base-port` 5016, `--genesis ../chain/genesis/acrux-net.json` | a twelve-validator sandbox for 12 s with the scenario's fault; writes `events.jsonl` and `outcome.json` into `--out` |
| `acrux identities` | `--out` required | writes the twelve validator identity keypairs derived from `ACRUX_IDENTITY_SEED`; prints each with its hub |

The registry id the hub commands default to is `ABjgHL8TcjxUjx6vfwsVQyvxiXR6pFgMkeESgwK6CbDm`. The six scenarios `--scenario` takes are `last-look`, `censorship`, `crash`, `equivocation`, `split-commitments` and `private-front-run`; the same seed gives the same schedule (Chapter 10).

Appendix C

# Hub Registry reference

The complete interface of `hub_registry`, program id `ABjgHL8TcjxUjx6vfwsVQyvxiXR6pFgMkeESgwK6CbDm`: every account with its byte layout, every instruction with its accounts and checks, every event and every error. Chapter 11 explains the design.

## C.1 On Solana mainnet

The program is live on Solana mainnet-beta. Its accounts and the values its Registry was initialized with are below; the API reads them, and `/v1/registry/config` answers `ready` on `mainnet-beta`.

Table C.1 The Hub Registry on Solana mainnet-beta.

| | |
|---|---|
| Program | `ABjgHL8TcjxUjx6vfwsVQyvxiXR6pFgMkeESgwK6CbDm` |
| ProgramData | `ATjqfhyJ53SAegfsDGixXxuDouTPQacT8T6YJUbMwpQA, upgradeable, deployed at slot 470,572,647` |
| Upgrade and config authority | `Gt5NYbCDCaJNjFmuX5CT3MNrREx1NVvNRbT4W8pZBv37` |
| Registry | `4eNA25M1QpgtumzFxVkfh8nDvUFVCzTW5jtbAd1KQvcF` |
| Fee vault | `Cugw8Vtd2TBPqVkVdc9LzrZm6DuzLB6z6ywhBCkrGkp5` |
| Fee destination | `G9Hx9dH8tNCKdL1FogM2Qsy493Hp6SrG6re1hxkq5aD6` |
| Values | deposit 1 SOL, registration fee 500 bps, cooldown 1 epoch, inactivity 4 epochs, $n_{min}$ 4 |
| Deploy | `29uy1NwjTLUW7YUjLE7RjUJhmiw1LpYhHFhc7qMgAzMenfxbLXjRQnvmVzhH2MoeyJCLeP2MhHN1KSgueuAe4gRr` |
| Initialize | `3ZQD9Mm1p4HStf3x3fgYoVNUYc1GYbRs4NnCKZ1QgvoahvP9rGHucDf2fWnKQJxbEsrY6eLw5NsVnza5gctsmZZh` |

## C.2 Encoding

Accounts and instruction data are Borsh-encoded by Anchor 0.31.1. An account starts with its 8-byte discriminator, then its fields in order, little endian, with no padding; an instruction's data starts with its 8-byte discriminator, then its arguments. A `pubkey` is 32 bytes; a `string` is a `u32` length and its bytes; an enum is one byte of variant index. Sizes below include the discriminator, and field offsets count from the account's first byte, so the first field is at offset 8. Epoch always means the Solana Clock epoch; rent comes from the Rent sysvar. The IDL is `backend/chain/target/idl/hub_registry.json`.

Table C.2 Program-derived addresses of `hub_registry`.

| Account | Seeds | Size | Count |
|---|---|---|---|
| Registry | `["registry"]` | 145 bytes | One |
| FeeVault | `["fee_vault"]` | 9 bytes | One |
| Hub | `["hub", code]` | 113 bytes | One per registered code |
| Membership | `["member", identity]` | 154 bytes | One per validator identity |
| Event authority | `["__event_authority"]` | | Signs the self-CPI events; holds no data |

## C.3 Accounts

### C.3.1 Registry

The configuration and the count of hubs ever registered. Written by `initialize`; `hubs_registered` is incremented by `register_hub`. Discriminator `2f ae 6e f6 b8 b6 fc da`.

Table C.3 Registry, 145 bytes.

| Offset | Field | Type | Size | Meaning |
|---|---|---|---|---|
| 8 | `authority` | `pubkey` | 32 | The upgrade authority that initialized the registry |
| 40 | `fee_destination` | `pubkey` | 32 | Where `sweep_fees` sends the fee vault's balance |
| 72 | `fee_vault` | `pubkey` | 32 | The FeeVault address |
| 104 | `deposit_lamports` | `u64` | 8 | The deposit a registration locks in the hub account |
| 112 | `registration_fee_bps` | `u16` | 2 | Registration fee, in basis points of the deposit |
| 114 | `cooldown_epochs` | `u64` | 8 | Epochs after a leave or move takes effect before the next change |
| 122 | `inactivity_epochs` | `u64` | 8 | A hub is releasable once the epoch is past its last active epoch by more than this |
| 130 | `min_members` | `u32` | 4 | Members a hub needs in an epoch to count as active ($n_{min}$) |
| 134 | `code_min_len` | `u8` | 1 | The shortest code accepted |
| 135 | `code_max_len` | `u8` | 1 | The longest code accepted |
| 136 | `hubs_registered` | `u64` | 8 | Hubs ever registered; the next hub's id is this plus one |
| 144 | `bump` | `u8` | 1 | The PDA bump |

### C.3.2 FeeVault

Registration fees sit in this account's lamports above its rent. No key signs for it; `sweep_fees` moves them to `fee_destination`. Discriminator `c0 b2 45 e8 3a 95 9d 84`; 9 bytes: the discriminator and `bump` (`u8`, offset 8).

### C.3.3 Hub

One registered hub. The deposit sits in the account's lamports above its rent. Discriminator `8f bd 04 78 a9 ea 0e 7d`.

Table C.4 Hub, 113 bytes.

| Offset | Field | Type | Size | Meaning |
|---|---|---|---|---|
| 8 | `id` | `u64` | 8 | Unique per registration; a code registered again after a release gets a new id |
| 16 | `code` | `[u8; 8]` | 8 | The code, padded with zero bytes |
| 24 | `location` | `Location` | 12 | ECEF `x`, `y`, `z` at offsets 24, 28, 32 |
| 36 | `registrant` | `pubkey` | 32 | Who registered the hub; receives the deposit back on release |
| 68 | `deposit_lamports` | `u64` | 8 | Held in this account above its rent |
| 76 | `members` | `u32` | 4 | Members in effect during `counted_epoch` |
| 80 | `joining` | `u32` | 4 | Joins that take effect at `counted_epoch + 1` |
| 84 | `leaving` | `u32` | 4 | Departures that take effect at `counted_epoch + 1` |
| 88 | `counted_epoch` | `u64` | 8 | The epoch the three counts were last brought up to |
| 96 | `created_epoch` | `u64` | 8 | The epoch the hub was registered |
| 104 | `last_active_epoch` | `u64` | 8 | The last epoch with at least `min_members` in effect, as of `counted_epoch`; equal to `created_epoch` for a hub never active |
| 112 | `bump` | `u8` | 1 | The PDA bump |

### C.3.4 Membership

One validator identity's membership, its pending change and its cooldown. Discriminator `e7 8d b4 62 6d a8 af a6`.

Table C.5 Membership, 154 bytes.

| Offset | Field | Type | Size | Meaning |
|---|---|---|---|---|
| 8 | `identity` | `pubkey` | 32 | The validator identity |
| 40 | `hub` | `pubkey` | 32 | The member's hub, or the default key when it has none |
| 72 | `hub_id` | `u64` | 8 | The `id` of `hub` when the member joined it |
| 80 | `joined_epoch` | `u64` | 8 | The epoch of the join or move that led to `hub` |
| 88 | `effective_epoch` | `u64` | 8 | The first epoch the membership in `hub` counts |
| 96 | `pending` | `Pending` | 1 | A leave or move waiting for `pending_epoch`; in effect from that epoch whether or not an instruction has written it into `hub` |
| 97 | `pending_hub` | `pubkey` | 32 | The hub of a pending move |
| 129 | `pending_hub_id` | `u64` | 8 | The `id` of `pending_hub` |
| 137 | `pending_epoch` | `u64` | 8 | The epoch the pending change takes effect |

| Offset | Field | Type | Size | Meaning |
|---|---|---|---|---|
| 145 | `cooldown_until_epoch` | `u64` | 8 | No join, leave or move before this epoch |
| 153 | `bump` | `u8` | 1 | The PDA bump |

### C.3.5 Types

Table C.6

The defined types: `Location` (12 bytes), `Pending` (1 byte) and `Config`, the argument of `initialize` (64 bytes).

| Type | Field or variant | Type or index | Meaning |
|---|---|---|---|
| `Location` | `x` | `i32` | Metres from the Earth's centre toward 0° latitude, 0° longitude (WGS84 ECEF) |
| | `y` | `i32` | Metres toward 0° latitude, 90° east |
| | `z` | `i32` | Metres toward the North Pole |
| `Pending` | `None` | `0` | No change waiting |
| | `Leave` | `1` | A leave takes effect at `pending_epoch` |
| | `Move` | `2` | A move to `pending_hub` takes effect at `pending_epoch` |
| `Config` | `fee_destination` | `pubkey` | Where `sweep_fees` sends the fee vault's balance |
| | `deposit_lamports` | `u64` | The deposit per hub, a configuration point |
| | `registration_fee_bps` | `u16` | The fee in basis points of the deposit, at most 10,000, a configuration point |
| | `cooldown_epochs` | `u64` | Epochs after a leave or move takes effect before the next change |
| | `inactivity_epochs` | `u64` | Epochs past its last active epoch after which a hub is releasable |
| | `min_members` | `u32` | Members a hub needs in an epoch to be active ($n_{min}$) |
| | `code_min_len` | `u8` | The shortest code accepted, at least 1 |
| | `code_max_len` | `u8` | The longest code accepted, at most 8 |

## C.4 Instructions

Every instruction but `initialize` emits its event by self CPI, so its account list ends with two accounts the tables below leave out: `event_authority` (PDA `["__event_authority"]`) and `program` (this program). In the account tables, W is writable and S is a signer. Checks are listed in the order the program makes them; Anchor's own account constraints (seeds, `has_one`, `address`, account creation) run first and fail with Anchor's codes.

### C.4.1 initialize(config: Config)

Discriminator `af af 6d 1f 0d 98 9b ed`. Creates Registry and FeeVault with the configuration and `hubs_registered` = 0.

Table C.7 Accounts of `initialize`.

| # | Account | W | S | Address and constraint |
|---|---|---|---|---|
| 0 | `authority` | W | S | The program's upgrade authority; pays for both accounts |
| 1 | `registry` | W | | PDA `["registry"]`, created |
| 2 | `fee_vault` | W | | PDA `["fee_vault"]`, created |
| 3 | `program` | | | This program; its ProgramData address must be `program_data` (`NotUpgradeAuthority`) |
| 4 | `program_data` | | | Its upgrade authority must be `authority` (`NotUpgradeAuthority`) |
| 5 | `system_program` | | | System program |

Checks: `registration_fee_bps` ≤ 10,000; 1 ≤ `code_min_len` ≤ `code_max_len` ≤ 8 (`InvalidConfig`). There is no instruction that changes the configuration afterwards.

### C.4.2 register_hub(code: string, location: Location)

Discriminator `6a 26 08 a8 ea 75 14 44`. Creates the hub and moves the deposit and the fee.

Table C.8 Accounts of `register_hub`.

| # | Account | W | S | Address and constraint |
|---|---|---|---|---|
| 0 | `registrant` | W | S | Pays the deposit, the fee and the rent |
| 1 | `registry` | W | | PDA `["registry"]`; `has_one = fee_vault` |
| 2 | `hub` | W | | PDA `["hub", code]`, created; fails if the code is taken |
| 3 | `fee_vault` | W | | The registry's fee vault; receives the fee |
| 4 | `system_program` | | | System program |

1. The code's length lies in `code_min_len..=code_max_len` and every byte is `A` to `Z` (`InvalidCode`).
2. $x^2 + y^2 + z^2$ lies in [6,300,000², 6,400,000²] metres squared (`InvalidLocation`).
3. `fee = deposit_lamports × registration_fee_bps / 10,000`, computed in 128 bits and rounded down; `hubs_registered` += 1.
4. The hub is written with `id = hubs_registered`, the code zero-padded, the location, the registrant, the deposit, zero counts, and `counted_epoch = created_epoch = last_active_epoch` = the current epoch.
5. System transfers from the registrant: the deposit to the hub, the fee to the fee vault. Emits `HubRegistered`.

### C.4.3 join_hub()

Discriminator `4e bd d6 1b 13 ad ce 67`. The validator identity joins a hub from the next epoch.

Table C.9 Accounts of `join_hub`.

| # | Account | W | S | Address and constraint |
|---|---|---|---|---|
| 0 | `payer` | W | S | Pays the membership's rent; may be the identity |
| 1 | `identity` | | S | The validator identity |
| 2 | `registry` | | | PDA `["registry"]` |
| 3 | `hub` | W | | The hub to join (a Hub account of this program) |
| 4 | `membership` | W | | PDA `["member", identity]`, created on the first join |
| 5 | `system_program` | | | System program |

1. Settles a leave or move whose `pending_epoch` has come into `hub`, `hub_id` and `effective_epoch`.
2. Nothing is pending ( `ChangePending` ); the identity is in no hub ( `AlreadyMember` ); the epoch is at least `cooldown_until_epoch` ( `CooldownActive` ).
3. Rolls the hub to the current epoch and adds 1 to `joining`.
4. Sets `hub`, `hub_id`, `joined_epoch` = the epoch, `effective_epoch` = the epoch + 1. A first join sets no cooldown. Emits `MemberJoined`.

### C.4.4 leave_hub()

Discriminator `87 4e fd b5 1d 2b f9 76`. The membership ends at the next epoch boundary.

Table C.10 Accounts of `leave_hub`.

| # | Account | W | S | Address and constraint |
|---|---|---|---|---|
| 0 | `identity` | | S | The validator identity |
| 1 | `registry` | | | PDA `["registry"]` |
| 2 | `hub` | W | | The member's hub address, checked against the membership; may have been released |
| 3 | `membership` | W | | PDA `["member", identity]` |

1. Settles any change that has come into effect. Nothing is pending ( `ChangePending` ); the identity is in a hub ( `NotAMember` ); `hub` is that hub ( `WrongHub` ); the epoch is at least `effective_epoch` ( `MembershipPending` ) and at least `cooldown_until_epoch` ( `CooldownActive` ).
2. Sets `pending` = Leave, `pending_epoch` = the epoch + 1, `cooldown_until_epoch` = `pending_epoch + cooldown_epochs`.
3. If the hub account still belongs to the program, holds data and has the member's `hub_id`, rolls it and adds 1 to `leaving`. Emits `MemberLeaving`.

### C.4.5 move_hub()

Discriminator `25 47 b0 2a 9b 8d 72 26`. The member changes hub at the next epoch boundary.

**Table C.11** Accounts of `move_hub`.

| # | Account | W | S | Address and constraint |
|---|---|---|---|---|
| 0 | `identity` | | S | The validator identity |
| 1 | `registry` | | | PDA `["registry"]` |
| 2 | `from_hub` | W | | The member's hub address, checked against the membership; may have been released |
| 3 | `to_hub` | W | | The hub to move to (a Hub account of this program) |
| 4 | `membership` | W | | PDA `["member", identity]` |

1. `to_hub` differs from `from_hub` (`SameHub`), then the checks of `leave_hub` on `from_hub`.
2. Sets `pending` = Move, `pending_hub` and `pending_hub_id` to the target, `pending_epoch` and `cooldown_until_epoch` as for a leave.
3. Records the departure on `from_hub` as a leave does; rolls `to_hub` and adds 1 to its `joining`. Emits `MemberMoving`. When the move settles, `joined_epoch` becomes `pending_epoch - 1`.

### C.4.6 release_hub()

Discriminator `5d c6 bc 5d 12 16 e1 24`. Closes a hub past its inactivity period. Anyone can send it.

**Table C.12** Accounts of `release_hub`.

| # | Account | W | S | Address and constraint |
|---|---|---|---|---|
| 0 | `registry` | | | PDA `["registry"]` |
| 1 | `hub` | W | | Closed to `registrant`; `has_one = registrant` |
| 2 | `registrant` | W | | The hub's registrant; receives the deposit and the rent |

Checks: rolls the hub to the current epoch, then requires the epoch to be greater than `last_active_epoch + inactivity_epochs` (`HubNotInactive`). Emits `HubReleased` with the lamports returned; the account closes, and the code is free to register again under a new id.

### C.4.7 sweep_fees()

Discriminator `af e1 62 47 76 42 22 94`. Sends the fee vault's lamports above its rent to the fee destination. Anyone can send it.

Table C.13 Accounts of `sweep_fees`.

| # | Account | W | S | Address and constraint |
|---|---|---|---|---|
| 0 | `registry` | | | PDA `["registry"]`; `has_one = fee_vault` |
| 1 | `fee_vault` | W | | The registry's fee vault |
| 2 | `destination` | W | | `address = registry.fee_destination` |

Checks: the vault's lamports minus its rent-exempt minimum is greater than zero (`NothingToSweep`). Emits `FeesSwept`, then moves the lamports directly from the vault to the destination.

## C.5 Reading the state at an epoch

A hub's count and a membership at any epoch *e* are known from their own accounts, without a crank. The API's indexer and any client apply these rules.

| | |
|---|---|
| **Members of a hub** | `members` when *e* = `counted_epoch`; `members + joining - leaving` when *e* > `counted_epoch`. |
| **A membership** | When `pending` is not None and *e* ≥ `pending_epoch`, the change is in effect: since `pending_epoch` the member is in `pending_hub` (a move) or in no hub (a leave). Otherwise it belongs to `hub`, counting from `effective_epoch`. |
| **A released hub** | A membership whose `hub_id` differs from the `id` of the account at `hub`, or whose hub account is gone, counts for no hub. |
| **Active** | At least `min_members` memberships in effect in the epoch. A hub never active has `last_active_epoch` = `created_epoch`. |
| **Releasable** | *e* > `last_active_epoch + inactivity_epochs`, with `last_active_epoch` rolled forward to *e* as the program does. |
| **Cooldown** | No join, leave or move before `cooldown_until_epoch`, which a leave or move sets to the epoch it takes effect plus `cooldown_epochs`. |

## C.6 Events

Events are emitted by self CPI, so they stay readable in transactions whose logs are truncated.

Table C.14 Events of `hub_registry`.

| Event | Fields | Emitted by |
|---|---|---|
| `HubRegistered` | `hub` pubkey, `id` u64, `code` [u8; 8], `location` Location, `registrant` pubkey, `deposit_lamports` u64, `fee_lamports` u64, `epoch` u64 | `register_hub` |
| `MemberJoined` | `identity` pubkey, `hub` pubkey, `hub_id` u64, `effective_epoch` u64 | `join_hub` |
| `MemberLeaving` | `identity` pubkey, `hub` pubkey, `hub_id` u64, `effective_epoch` u64, `cooldown_until_epoch` u64 | `leave_hub` |
| `MemberMoving` | `identity` pubkey, `from_hub` pubkey, `from_hub_id` u64, `to_hub` pubkey, `to_hub_id` u64, `effective_epoch` u64, `cooldown_until_epoch` u64 | `move_hub` |
| `HubReleased` | `hub` pubkey, `id` u64, `code` [u8; 8], `registrant` pubkey, `returned_lamports` u64, `epoch` u64 | `release_hub` |
| `FeesSwept` | `destination` pubkey, `lamports` u64 | `sweep_fees` |

## C.7 Errors

Anchor custom errors, returned as `custom program error: 0x…` with the code below.

Table C.15 Error codes of `hub_registry`.

| Code | Name | Message | Raised by |
|---|---|---|---|
| 6000 | `NotUpgradeAuthority` | Only the program's upgrade authority can initialize the registry | `initialize` |
| 6001 | `InvalidConfig` | The configuration is out of range | `initialize` |
| 6002 | `InvalidCode` | A hub code is capital letters A–Z within the registry's length range | `register_hub` |
| 6003 | `InvalidLocation` | The location is not an ECEF point near the Earth's surface, in metres | `register_hub` |
| 6004 | `AlreadyMember` | The validator is already a member of a hub | `join_hub` |
| 6005 | `NotAMember` | The validator is not a member of a hub | `leave_hub, move_hub` |
| 6006 | `WrongHub` | The hub account is not the member's hub | `leave_hub, move_hub` |
| 6007 | `SameHub` | The validator is already in that hub | `move_hub` |
| 6008 | `MembershipPending` | The membership takes effect at the next epoch boundary | `leave_hub, move_hub` |
| 6009 | `ChangePending` | A leave or move is already waiting for the next epoch boundary | `join_hub, leave_hub, move_hub` |
| 6010 | `CooldownActive` | The validator's cooldown has not ended | `join_hub, leave_hub, move_hub` |
| 6011 | `HubNotInactive` | The hub has not been inactive for the registry's period | `release_hub` |

| Code | Name | Message | Raised by |
|---|---|---|---|
| 6012 | `NothingToSweep` | The fee vault holds nothing above its rent | `sweep_fees` |

Appendix D

# Wire formats

Every byte acrux-node sends between validators and every byte it signs, as `backend/node/crates/acrux-node/src/types.rs`, `acrux-crypto` and `acrux-execution/src/threshold.rs` define them. The site's reference page, `web/content/docs/reference/wire-formats.mdx`, states the same layouts.

## D.1 Encoding

Node-to-node messages travel over QUIC, one message per unidirectional stream, at most 1 MiB, encoded with bincode 1.3 under its default options. The rules below fix every layout in this appendix.

Table D.1 bincode 1.3 default encoding, as acrux-node writes it with `bincode::serialize`.

| Type | Bytes |
|---|---|
| `u8, bool` | 1 byte; `bool` is 0 or 1 |
| `u32, u64` | 4 or 8 bytes, little-endian, fixed width |
| `usize` | as `u64`, 8 bytes |
| `[u8; N]` | N bytes, no length prefix |
| `Vec<T>, String` | the length as `u64`, then the elements (UTF-8 bytes for a string) |
| `Option<T>` | one byte, 0 for none or 1 for some, then the value if some |
| `enum` | the variant index as `u32`, then the variant's fields |
| `struct, tuple` | the fields in declaration order, no tags or padding |

A receiver decodes with the same fixed-width options, a limit of 1 MiB and no trailing bytes allowed (`protocol.rs`, `decode`). Hashes are SHA-256, 32 bytes. Signatures are Ed25519, 64 bytes, carried as a `Vec<u8>` and checked with strict verification. The JSON-RPC and the event stream carry the same objects as JSON, hashes in lowercase hex and keys and signatures in base58 (Appendix B).

## D.2 The envelope

Every packet is one `Envelope`. Its signature is the sender's Ed25519 signature over the bincode bytes of `body`, its variant index included. A commitment's or an attestation's signature is this envelope signature: the same bytes are what a proposer signs and what a slashing proof carries.

Table D.2 `Envelope`, the packet.

| Offset | Size | Field | Type | Meaning |
|---|---|---|---|---|
| 0 | 8 | `from` | `usize` | the sender's validator index, 0 to 11 on Acrux Net |
| 8 | 4 | `body tag` | `u32` | the `Wire` variant, Table D.3 |
| 12 | B | `body` | `Wire` | the variant's fields |
| 12 + B | 8 | `signature len` | `u64` | 64 |
| 20 + B | 64 | `signature` | `Vec<u8>` | Ed25519 over the bytes at offsets 8 to 12 + B |

Because `from` has a fixed width, bytes 8 to 12 of every packet name its variant. The ingress task reads them to send commitments down their own path before decoding anything (Section 8.2.2).

Table D.3 The `Wire` variants.

| Tag | Variant | Fields | Sent |
|---|---|---|---|
| 0 | `Shreds` | `Vec<Shred>` | proposer to the relay sampled for each shred |
| 1 | `Relay` | `Vec<Shred>` | relay to all validators |
| 2 | `Commit` | `Commitment` | proposer to its co-proposers, three times |
| 3 | `Attest` | `Attestation` | proposer to all validators |
| 4 | `Vote` | `SignedVote` | validator to all validators |
| 5 | `Certificate` | `certificate: Certificate, proofs: Vec<SignedVote>` | validator to all validators |
| 6 | `Submit` | `ValidatedTx` | gateway or previous proposer to a proposer |
| 7 | `SubmitFragment` | `VerifiedFragment` | gateway or previous proposer to the receiving proposer |
| 8 | `Handover` | `slot: u64, parent: BlockRef, shreds: Vec<Shred>, attestations: Vec<(Attestation, Vec<u8>)>` | a window's proposers to the next window's proposers |
| 9 | `Equivocation` | `first, second: (Commitment, Vec<u8>)` | any validator to all validators |

`BlockRef` is `slot: u64` then `hash: [u8; 32]`, 40 bytes. A `Handover`'s attestations each carry the attesting proposer's signature over `Wire::Attest` of that attestation, so the next proposers can check them without the original envelopes.

## D.3 Commitment and attestation

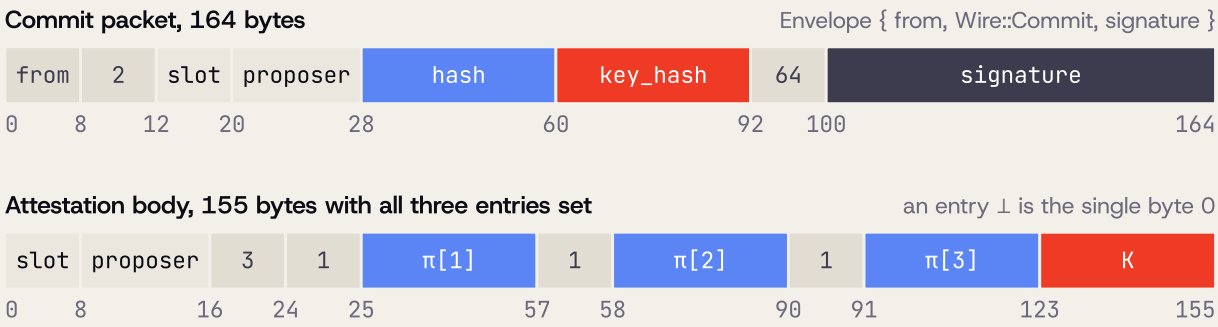



**Figure D.1** Byte layouts of a commitment packet and an attestation body. Blue fields are hashes, red the block key or its hash, dark the signature; the figures under each bar are byte offsets. Fields of a few bytes are widened for their labels.

**Table D.4** `Commitment`, 80 bytes. In a packet it starts at offset 12.

| Offset | Size | Field | Type | Meaning |
|---|---|---|---|---|
| 0 | 8 | `slot` | `u64` | the slot |
| 8 | 8 | `proposer` | `usize` | the proposer's role, 1 to 3 |
| 16 | 32 | `hash` | `[u8; 32]` | SHA-256 of the bincode bytes of the `BlockCommitment` |
| 48 | 32 | `key_hash` | `[u8; 32]` | SHA-256 of the block's 32-byte key |

**Table D.5** `BlockCommitment`, the object `hash` covers. It is never sent; each validator rebuilds it from the shreds it holds.

| Offset | Size | Field | Type | Meaning |
|---|---|---|---|---|
| 0 | 8 | `slot` | `u64` | the slot |
| 8 | 8 | `proposer` | `usize` | the proposer's role |
| 16 | 32 | `key_hash` | `[u8; 32]` | SHA-256 of the block key |
| 48 | 1 or 41 | `parent` | `Option<BlockRef>` | the cblock the window builds on; some only in the first slot of a window |
| 49 or 89 | 8 + 32*n* | `roots` | `Vec<[u8; 32]>` | the Merkle root of each of the block's *n* slices, in slice order |

The commitment binds every slice's ciphertext through its root and the key through its hash, so a key released later opens exactly the committed block. Two commitments signed by one proposer for one slot with different `hash` are a slashing proof, and `Wire::Equivocation` carries both with their signatures.

Table D.6 `Attestation`, 155 bytes when every entry of $\pi$ is set; each $\perp$ entry is one byte instead of 33.

| Offset | Size | Field | Type | Meaning |
|---|---|---|---|---|
| 0 | 8 | `slot` | `u64` | the slot |
| 8 | 8 | `proposer` | `usize` | the attesting proposer's role |
| 16 | 8 | `pi len` | `u64` | 3 |
| 24 | 3 to 99 | `pi` | `Vec<Option<[u8; 32]>>` | entry $q - 1$ is proposer $q$'s commitment `hash` as received on time, or none ($\perp$) |
| 27 to 123 | 32 | `key` | `[u8; 32]` | the attesting proposer's block key |

A validator checks each set entry of $\pi$ against the `BlockCommitment` it rebuilds from that proposer's shreds, and SHA-256 of `key` against the `key_hash` in the shreds, before it decrypts.

## D.4 Slice encryption

Each slice of a block is encrypted with ChaCha20-Poly1305 under the block's fresh 32-byte key (`acrux-crypto`, `encrypt_slice`). The proposer cuts its block payload into chunks of 38,000 bytes, one per slice.

Table D.7 The inputs of a slice's AEAD. All integers are little-endian.

| Input | Size | Bytes |
|---|---|---|
| key | 32 | the block key $K$, from the operating system's generator |
| nonce | 12 | the first 12 bytes of SHA-256(`"acrux-slice-nonce-v1"` ‖ `slot` u64 ‖ `proposer` u64 ‖ `slice` u32), over 40 bytes |
| associated data | 34 | `"acrux-slice-v1"` ‖ `slot` u64 ‖ `proposer` u64 ‖ `slice` u32 |
| plaintext | ≤ 65,520 | a chunk of the bincode `BlockPayload` (`transactions: Vec<ValidatedTx>`, `fragments: Vec<VerifiedFragment>`) |
| ciphertext | ≤ 65,536 | the plaintext's length plus the 16-byte Poly1305 tag; 38,016 bytes for a full chunk |

## D.5 Shreds and their Merkle proofs

**Coding.** The ciphertext of length $L$ is cut into 32 data shards of $d = \lceil L/32 \rceil$ bytes, the last zero-padded, and Reed-Solomon over GF($2^8$) adds 32 coding shards, 64 in all. Any 32 distinct shards rebuild the codeword; the ciphertext is the first $L$ bytes of the 32 data shards. A full slice has $d = 1{,}188$.

**Merkle tree.** A leaf is SHA-256( `0x00` ‖ `index` u8 ‖ shard), an inner node SHA-256( `0x01` ‖ left ‖ right). The 64 leaves give a tree of depth 6 and one root per slice. A shred's proof is its six siblings from the leaf up; at each level an even index hashes as the left child.

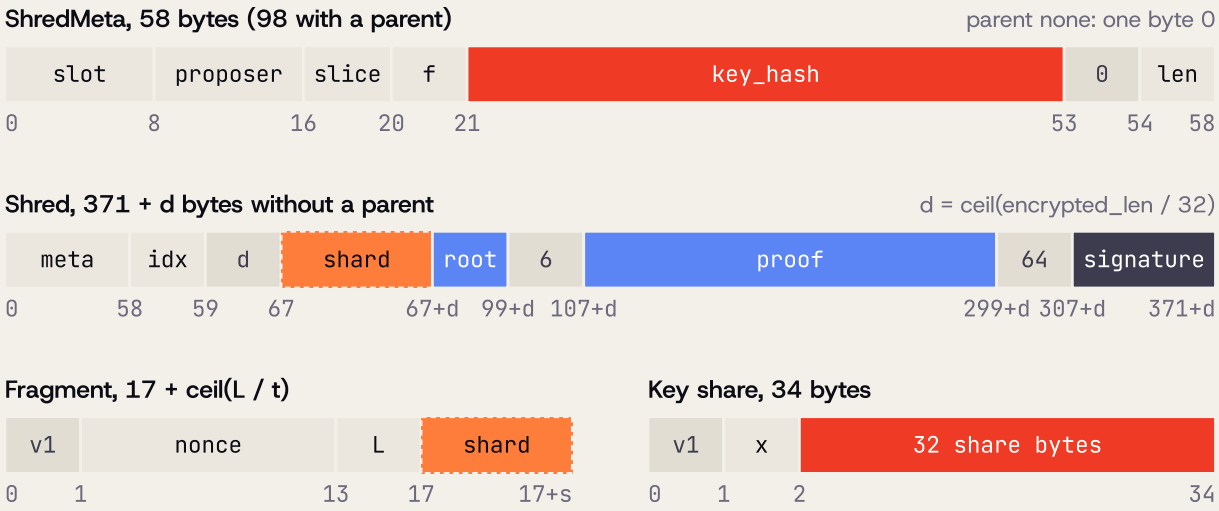



**Figure D.2** Byte layouts of a shred's metadata and of a shred, and of a threshold fragment and its key share after base64 decoding. Orange fields have a variable length: *d* for a shred's shard, *s* = ⌈*L*/*t*⌉ for a fragment's.

**Table D.8** `ShredMeta`, identical in every shred of a slice: 58 bytes, or 98 with a parent.

| Offset | Size | Field | Type | Meaning |
|---|---|---|---|---|
| 0 | 8 | `slot` | `u64` | the slot |
| 8 | 8 | `proposer` | `usize` | the proposer's role |
| 16 | 4 | `slice` | `u32` | the slice's index in the block |
| 20 | 1 | `final_slice` | `bool` | the block's last slice |
| 21 | 32 | `key_hash` | `[u8; 32]` | SHA-256 of the block key |
| 53 | 1 or 41 | `parent` | `Option<(u64, [u8; 32])>` | the parent cblock's slot and hash, set only in the final slice of a window's first block |
| 54 or 94 | 4 | `encrypted_len` | `u32` | the slice ciphertext's length *L*, 1 to 65,536 |

Table D.9 `Shred`, offsets for metadata without a parent (add 40 with one): 371 + *d* bytes, 1,559 for a full slice.

| Offset | Size | Field | Type | Meaning |
|---|---|---|---|---|
| 0 | 58 | `meta` | `ShredMeta` | above |
| 58 | 1 | `index` | `u8` | the shard's position, 0 to 63; 0 to 31 are data |
| 59 | 8 + *d* | `data` | `Vec<u8>` | the shard |
| 67 + *d* | 32 | `root` | `[u8; 32]` | the slice's Merkle root |
| 99 + *d* | 200 | `proof` | `Vec<[u8; 32]>` | length 6, then the six siblings from the leaf up |
| 299 + *d* | 72 | `signature` | `Vec<u8>` | the proposer's signature, below |

The shred signature covers the bincode bytes of the tuple ( `"acrux-shred-v1"` as 14 fixed bytes, `meta`, `index`, `root`, SHA-256( `data` )): 137 bytes, or 177 with a parent. It binds the slice's finality, parent and length as well as the indexed root. A receiver checks the index, the length, the shard size against `encrypted_len`, the proof depth of 6, the signature and the proof against `root` before it keeps the shred ( `verify_shred` ); after rebuilding, it recomputes the root from all 64 shards.

## D.6 Votes and certificates

Table D.10 `Vote`, 53 bytes, or 21 for a vote with no hash. `SignedVote` is the vote followed by the voter's signature over its bincode bytes as a `Vec<u8>`.

| Offset | Size | Field | Type | Meaning |
|---|---|---|---|---|
| 0 | 8 | `validator` | `usize` | the voter's index |
| 8 | 8 | `slot` | `u64` | the consensus slot |
| 16 | 1 or 33 | `hash` | `Option<[u8; 32]>` | the cblock hash voted for; none for skip votes |
| 49 or 17 | 4 | `kind` | `VoteKind` | `Notar` 0, `NotarFallback` 1, `Skip` 2, `SkipFallback` 3, `Finalize` 4 |

Table D.11 `Certificate`. In `Wire::Certificate` it is followed by `proofs`, the `SignedVote` of each vote it counts.

| Field | Type | Meaning |
|---|---|---|
| `kind` | `CertificateKind` | `Notarization` 0, `NotarFallback` 1, `FastFinalization` 2, `Finalization` 3, `Skip` 4 |
| `slot` | `u64` | the consensus slot |
| `hash` | `Option<[u8; 32]>` | the certified cblock; none for a skip certificate |
| `votes` | `Vec<Vote>` | the votes whose distinct stake meets the threshold: 60%, or 80% of initial notar votes for fast finalization |

A receiver checks every proof's signature and that the proofs are exactly the certificate's votes before `Votor::receive` counts it. The cblock hash a notar vote names is SHA-256 of the bincode tuple ( `i` u64, `parent` `Option<BlockRef>`, the four slots as `Vec<(u64, Option<Vec<usize>>, Vec<Option<[u8; 32]>>)>` of slot, agreed set and $\pi$, and the state hash as a hex `String` ).

## D.7 Transactions

`Wire::Submit` carries a `ValidatedTx`: `bytes` as a `Vec<u8>` (the signed Solana transaction, at most 1,232 bytes), its `TxHeader`, and `signature_count` as `u64`. In bincode the header is `id`, `fee_payer` and `priority_fee_per_cu` as strings, `compute_limit` as `u64`, `writable_accounts` as a vector of strings, and `target_slot` as `u64`. Every receiver re-derives the header from the bytes and drops a copy whose header differs. The id is SHA-256 of the serialized message, in lowercase hex.

## D.8 Threshold fragment

A $t$-of-$n$ transaction travels as one `FragmentSubmission` per receiver through `submitFragment`, with $2 \le t \le n \le 3$. `@acrux/sdk` writes these bytes and `acrux-execution/src/threshold.rs` reads them; `seal_for_receivers` is the reference sealer.

- **Header bytes.** The UTF-8 bytes of the `TxHeader` as compact JSON, fields in exactly this order: `id`, `feePayer`, `priorityFeePerCu`, `computeLimit`, `writableAccounts`, `targetSlot`, with no whitespace. The fee payer signs these bytes, and they are the ciphertext's associated data.
- **Ciphertext.** ChaCha20-Poly1305 of the signed transaction's bytes under a fresh 32-byte key and a random 12-byte nonce; its length $L$ is 16 to 1,248 bytes.
- **Shards.** The ciphertext is cut into $t$ data shards of $\lceil L/t \rceil$ bytes, the last zero-padded, and extended to $n$ with systematic Reed-Solomon over GF($2^8$) reduced by 0x11d. Fragment $i$ carries shard $i$.
- **Key shares.** Each key byte is the constant term of a random polynomial of degree $t - 1$ over GF($2^8$) reduced by 0x11b; share $i$ is the 32 polynomials' values at $x = i + 1$.

**Table D.12** The decoded `fragment` and `keyShare` envelopes, version 1.

| Envelope | Offset | Size | Field | Meaning |
|---|---|---|---|---|
| `fragment` | 0 | 1 | `version` | 0x01 |
| | 1 | 12 | `nonce` | the ciphertext's nonce |
| | 13 | 4 | `length` | $L$ as `u32`, little-endian |
| | 17 | $\lceil L/t \rceil$ | `shard` | shard `index` |
| `keyShare` | 0 | 1 | `version` | 0x01 |
| | 1 | 1 | `x` | `index` + 1 |
| | 2 | 32 | `share` | the key share's 32 bytes |

**Table D.13** `FragmentSubmission`, the JSON object one receiver gets. Unknown fields are refused.

| Field | Encoding | Meaning |
|---|---|---|
| `header` | TxHeader as JSON | the public header the fee payer signed |
| `n, t` | integers | fragments in all and fragments to decode |
| `index` | integer | the fragment's index, 0 to $n - 1$ |
| `receiver` | integer | the receiving proposer's role, 1 to 3 |
| `fragment` | base64, at most 2,048 characters | the fragment envelope |
| `keyShare` | base64, at most 48 characters | the key-share envelope, 34 bytes |
| `headerSignature` | base58 | the fee payer's signature over the header bytes |

Every fragment of one id carries the same header, $n$, $t$, nonce and length, and a receiver takes one fragment per transaction. On the wire between validators a fragment travels decoded, as `VerifiedFragment` in bincode (`header`, `n`, `t`, `index`, `receiver` as `u8`s, `nonce` [u8; 12], `ciphertext_len` u32, `shard`, `share` [u8; 32], `header_signature`), and every receiver re-encodes and re-verifies it with `revalidate`. Validators rebuild the transaction in the slot that accepts the $t$-th distinct fragment, by inverting the chosen rows of the coding matrix and interpolating the key at $x = 0$, then check that the decoded transaction derives the signed header.

## D.9 Hashed objects

Table D.14 Bytes that are never sent but hashed with SHA-256, which every validator must produce identically.

| Value | SHA-256 of | Code |
|---|---|---|
| Block commitment | bincode `BlockCommitment` | `begin_block`, `try_execute` |
| Account hash | `"acrux-account-v1"` ‖ address ‖ lamports u64 ‖ owner ‖ executable u8 ‖ rent_epoch u64 ‖ data length u64 ‖ data | `SvmExecutor::refresh` |
| State hash | not a hash itself: the account hashes summed as 256-bit little-endian integers mod $2^{256}$, in hex | `SvmExecutor::state_hash` |
| Blockhash | previous blockhash (32 zero bytes at genesis) ‖ state accumulator ‖ slot u64 | `advance_blockhash` |
| Cblock hash | bincode ( `i` , `parent` , [(slot, P, $\pi$)], state hash) | `try_vote` |
| Transaction id | the serialized Solana message | `validate_transaction` |

Appendix E

# From the paper to the code

Each line of the paper beside the code that runs it. The tables set `backend/spec/map.json` in full, one section per algorithm as the site's `/spec` page shows it, then the validity and agreement conditions, the theorem and lemmas with their Fault Lab scenarios, and the sections the paper leaves to the implementation.

## E.1 Reading the tables

`map.json` records, for each reference in New Constellation v1.0 [1], the line range, the file, the function and the Fault Lab scenario that exercises it. The API serves it at `/v1/spec` and checks it against `@acrux/contract`'s `SpecMap` schema. Files are given relative to `backend/node/crates/`; a function written `Type::method` is a method, and several functions in one cell share the lines between them. The paper column says in a few words what the lines do; the algorithms themselves are set in Chapter 3, Chapter 4, Chapter 5 and Chapter 6, and the scenarios in Chapter 10.

## E.2 Algorithm 1

Proposer $p$ of the window starting with consensus slot $i_0$. Implemented in `acrux-node`: the slot clock in `protocol.rs`, the hash exchange in `exchange.rs`.

Table E.1 Algorithm 1, line by line.

| Lines | The paper | Function | File | Scenario |
|---|---|---|---|---|
| 1-6 | the window's proposers and state; on ParentReady($i_0$) or executing slot $i_0 - 1$, run the window's slots back to back | `Network::start_window`, `Network::tick` | `acrux-node/src/protocol.rs` | Crash |
| 7-9 | fresh key $K_s$, pack and encrypt $b$, stream its slices through Rotor | `Network::begin_block` | `acrux-node/src/protocol.rs` | Last look |
| 10-13 | first slot of the window: wait for ParentReady, put the parent in the last slice | `Network::prepare_early`, `Network::complete_early` | `acrux-node/src/protocol.rs` | |
| 14-16 | start[$s$], proposal[$s$][$p$] ← h($b$), send the signed Commit a few times | `ExchangeEngine::launch` | `acrux-node/src/exchange.rs` | Last look |

| Lines | The paper | Function | File | Scenario |
|---|---|---|---|---|
| 17 | last slot of the window: send $b$ directly to the next proposers | `Network::try_vote`, `Network::receive`, `Wire::Handover` | `acrux-node/src/protocol.rs` | |
| 18-19 | on to the next slot without waiting for votes | `Network::tick` | `acrux-node/src/protocol.rs` | |
| 20-23 | first Commit from a co-proposer $q$: start[$s$], received[$s$][$q$], proposal[$s$][$q$] | `ExchangeEngine::receive_commit` | `acrux-node/src/exchange.rs` | Split commitments |
| 24-32 | all commitments in or $2\delta_{\text{local}}$ passed: build $\pi$, send the signed Attestation($s$, $p$, $\pi$, $K_s$) | `ExchangeEngine::launch`, `ExchangeState::attestation_vector` | `acrux-node/src/exchange.rs` | Last look |

## E.3 Algorithm 2

The validator. Alpenglow runs unchanged on consensus slots and cblocks; its timeouts and Block events come from this algorithm, in `protocol.rs` and `agreement.rs`.

**Table E.2** Algorithm 2, line by line.

| Lines | The paper | Function | File | Scenario |
|---|---|---|---|---|
| 1-8 | state; on ParentReady set Timeout($i$) and Deadline($i_0$) | `LocalSlot`, `WindowRun`, `Network::start_window`, `Network::try_vote` | `acrux-node/src/protocol.rs` | Crash |
| 9-10 | at Deadline($i_0$) stop waiting for attestations and TryVote | `Network::tick`, `Network::try_vote` | `acrux-node/src/protocol.rs` | Crash |
| 11-16 | a new shred: record equivocation evidence, reconstruct $b$[$s$][$q$], TryExecute | `Network::receive`, `Wire::Relay` | `acrux-node/src/protocol.rs` | Equivocation |
| 17-23 | an attestation: keep it or flag equivocation, TryExecute, TryVote | `Network::receive`, `Wire::Attest` | `acrux-node/src/protocol.rs` | Split commitments |
| 24-30 | TryExecute: Agreed($s$), all blocks held, decrypt under the released keys, hashes match $\pi$ | `Network::try_execute` | `acrux-node/src/protocol.rs` | Censorship |
| 31-37 | the parent: most common among the agreed blocks, ties to the lowest $q$; the parent's state | `Network::try_execute` | `acrux-node/src/protocol.rs` | Crash |
| 38-42 | execute Merge($s$), P[$s$] ← A, TryVote, TryExecute($s$ + 1) | `SvmExecutor::execute_slot` | `acrux-execution/src/execution.rs` | Censorship |

| Lines | The paper | Function | File | Scenario |
|---|---|---|---|---|
| 43-52 | TryVote: in order, wait at a window's first cblock, emit Block($i$, h($c_i$), par[$i$]) | `Network::try_vote` | `acrux-node/src/protocol.rs` | Crash |
| 53-59 | Agreed(s): one $\pi$, a strict majority, every attested proposer's attestation held, no equivocation | `Network::try_execute`, `Network::try_vote` | `acrux-node/src/protocol.rs` | Split commitments |
| 53-59 | Agreed(s): one $\pi$, a strict majority, every attested proposer's attestation held, no equivocation | `agreed` | `acrux-node/src/agreement.rs` | Split commitments |
| 24-42 | TryExecute as a follower runs it: decrypt the agreed blocks under the released keys, merge, re-execute, compare the state hash | `Replay::apply` (`acrux node follow`) | `acrux-node/src/cli.rs` | |

## E.4 Algorithm 3

The proposer schedule of an epoch with $N$ windows, in `acrux-schedule`. The API's schedule preview runs the same function through `acrux schedule`.

Table E.3 Algorithm 3, line by line.

| Lines | The paper | Function | File | Scenario |
|---|---|---|---|---|
| 1-4 | hub probabilities $\alpha$-weighted by stake, a hub drawn for each of the $N$ windows | `build_schedule` | `acrux-schedule/src/lib.rs` | |
| 5 | reorder the hub sequence into SIMD-0675 regional runs | `build_schedule regional grouping` | `acrux-schedule/src/lib.rs` | |
| 6-15 | per window, systematic sampling of $k$ proposers over the shuffled, capped member stakes | `build_schedule` | `acrux-schedule/src/lib.rs` | |

## E.5 Algorithm 4

Transaction selection on $m$ = 4 tracks of 2,000,000 CU, in `acrux-execution`.

Table E.4 Algorithm 4, line by line.

| Lines | The paper | Function | File | Scenario |
|---|---|---|---|---|
| 1-8 | SelectTxs: free spaces on $m$ tracks of $b_{cu}$, transactions sorted by priority | `sort_transactions`, `select_with_fees` | `acrux-execution/src/selection.rs` | Censorship |
| 9-11 | charge the priority fee; a transaction that cannot pay is skipped | `SvmExecutor::execute_slot` | `acrux-execution/src/execution.rs` | Censorship |
| 12-29 | place each transaction at the earliest free position whose writable accounts do not collide | `select_with_fees` | `acrux-execution/src/selection.rs` | Censorship |
| 30-37 | charge the inclusion fee; the selected list in priority order | `SvmExecutor::charge_inclusion`, `SvmExecutor::execute_slot` | `acrux-execution/src/execution.rs` | Censorship |
| 38-50 | FindSpace: split a free space around a placed transaction | `select_with_fees` | `acrux-execution/src/selection.rs` | Censorship |

## E.6 Validity and agreement

The paper states when a proposer block is valid, conditions (i) to (vii), and when a validator considers a slot agreed, conditions (a) to (d). Each is checked in one place in the code.

Table E.5 The validity conditions of a proposer block and the code that checks each one.

| Condition | The paper | Function | File |
|---|---|---|---|
| (i) | all shreds valid, signed by the proposer, Merkle roots match and reproduce the root | `authenticate`, `verify_shred`, `reconstruct_verified_slice` | `acrux-node/src/protocol.rs`, `acrux-crypto/src/lib.rs` |
| (ii) | the block is complete and received before its consensus slot's timeout | `Network::try_execute`, `Network::try_vote` | `acrux-node/src/protocol.rs` |
| (iii) | the only block of the proposer for the slot | `Network::receive` (`Wire::Relay`), `ExchangeEngine::receive_commit` | `acrux-node/src/protocol.rs`, `exchange.rs` |
| (iv) | the block hashes match the ones the attestations attest to | `Network::try_execute` (`BlockCommitment` against $\pi$) | `acrux-node/src/protocol.rs` |
| (v) | the payload decrypts under the attested key, and the key matches the commitment | `decrypt_slice`, $H(K)$ against `key_hash` | `acrux-crypto/src/lib.rs`, `acrux-node/src/protocol.rs` |
| (vi) | a window's first block carries a correct parent reference | `Network::try_execute`, `Votor::local_block` | `acrux-node/src/protocol.rs`, `acrux-consensus/src/lib.rs` |
| (vii) | size, format and compute limits | `SvmExecutor::validate_transaction`, `FragmentSubmission::verify`, `decode` | `acrux-execution/src/execution.rs`, `threshold.rs` |

Table E.6 The agreement conditions, all checked by `agreement::agreed` in `acrux-node/src/agreement.rs`.

| Condition | The paper | In `agreed` |
|---|---|---|
| (a) | all attestations received for the slot agree, and $P_s$ is the set they attest | every `pi` equals the first; the accepted set is the indices of its set entries |
| (b) | the attestation of every proposer in $P_s$ is held | each accepted proposer is among the observed attesters |
| (c) | $P_s$ is a strict majority, $\|P_s\| > k/2$ | more than one of three accepted |
| (d) | no equivocation of any proposer in slot $s$ | the slot's equivocation flag is clear, and no proposer signed two different attestations |

## E.7 Theorem and lemmas

The correctness results of the paper's Section 2.3, each beside the code path it rests on and the Fault Lab scenario that runs it on twelve validators. The proofs are in Chapter 7.

Table E.7 Theorem 2.1 and Lemmas 2.2 to 2.4.

| Result | Function | File | Scenario |
|---|---|---|---|
| Theorem 2.1 NewCon correctness | `Votor::drive`, `Votor::ready_parent` | `acrux-consensus/src/lib.rs` | Crash |
| Lemma 2.2 Selective-censorship resistance | `Network::try_execute`, `Network::try_vote` | `acrux-node/src/protocol.rs` | Censorship |
| Lemma 2.3 Proposal hiding | `encrypt_slice`, `decrypt_slice` | `acrux-crypto/src/lib.rs` | Last look |
| Lemma 2.4 Accountable equivocation | `Network::receive` | `acrux-node/src/protocol.rs` | Equivocation |
| Lemma 2.4 Accountable equivocation | `ExchangeEngine::receive_commit` (two signed commitments become a `Wire::Equivocation` proof) | `acrux-node/src/exchange.rs` | Equivocation |

## E.8 Threshold transactions, fees and the Fault Lab

Sections 3.2 and 3.4 of the paper describe threshold transactions and fees as directions an implementation fixes; acrux-node's choices are in Chapter 5. The Fault Lab entries are the runner that executes every scenario above.

**Table E.8** Sections 3.2 and 3.4 and the Fault Lab runner.

| Reference | Function | File | Scenario |
|---|---|---|---|
| Section 3.2 Threshold transactions | `FragmentSubmission::verify`, `ThresholdCache::accept_at`, `validate_decoded`, `Network::submit_fragment` | `acrux-execution/src/threshold.rs` | Private front-run |
| Section 3.4 Fees | `SvmExecutor::charge_inclusion`, `SvmExecutor::execute_slot` | `acrux-execution/src/execution.rs` | |
| Fault Lab Scenario runner | `run` (sandbox cluster, events.jsonl, outcome.json) | `acrux-node/src/lab.rs` | |
| Fault Lab Scenario runner | `Network::lab_inject`, `Network::lab_conclude` | `acrux-node/src/protocol.rs` | |
| Section 3.4 Fees | `SvmExecutor::advance_blockhash` (recent blockhashes, replay window) | `acrux-execution/src/execution.rs` | |

# References


[1] Quentin Kniep, Roger Wattenhofer. *New Constellation: Multiple Concurrent Proposers for Alpenglow*, v1.0. Solana Foundation, 2026-11-14.
https://github.com/wattenhoferSOL/SolanaMCP/blob/main/New%20Constellation%20v1.0.pdf

[2] Quentin Kniep, Max Resnick, Jakub Sliwinski, Roger Wattenhofer. *Solana Constellation: Internet Capital Markets*, white paper v0.9. Anza, 2026-03.
https://anza.xyz/constellationwhitepaper

[3] Quentin Kniep, Jakub Sliwinski, Roger Wattenhofer. *Solana Alpenglow Consensus: Increased Bandwidth, Reduced Latency*. Anza, 2025.
https://anza.xyz/alpenglow-1-1

[4] Solana Foundation. *Alpenglow: Live on Mainnet*. Activated 2026-10-21, epoch 1064.
https://solana.com/upgrades/alpenglow

[5] Pranav Garimidi, Joachim Neu, Max Resnick. *Multiple Concurrent Proposers: Why and How*. 2025. arXiv:2509.23984.

[6] Eric Budish, Peter Cramton, John Shim. *The High-Frequency Trading Arms Race: Frequent Batch Auctions as a Market Design Response*. Quarterly Journal of Economics 130(4), 2015.

[7] Philip Daian et al. *Flash Boys 2.0: Frontrunning in Decentralized Exchanges, Miner Extractable Value, and Consensus Instability*. IEEE Symposium on Security and Privacy, 2020.

[8] Gerzon, Weintraub, In, Mislove, Nita-Rotaru. *Quantifying the Threat of Sandwiching MEV on Jito*. ACM Internet Measurement Conference (IMC '25), 2025-10.
https://cnitarot.github.io/papers/imc26_solana.pdf

[9] QuickNode. *Solana MEV economics: Jito bundles and liquid staking*.
https://www.quicknode.com/blog/solana-mev-economics-jito-bundles-liquid-staking-guide

[10] Anza. *Constellation*. 2026.
https://www.anza.xyz/blog/constellation

[11] Hugo Krawczyk. *Secret Sharing Made Short*. CRYPTO '93, LNCS 773, 1994.

[12] Quentin Kniep, Roger Wattenhofer. *SIMD-0675: Geo-Aware Leader Schedule*. 2026.
https://github.com/solana-foundation/solana-improvement-documents/pull/675

[13] Solana mainnet-beta, `getRecentPerformanceSamples`: ten samples on 2026-11-17 (UTC), 212 to 218 ms per slot.

[14] Quentin Kniep, Roger Wattenhofer. *SIMD-0674: Validator Location Registration*. 2026.
https://github.com/solana-foundation/solana-improvement-documents/pull/674

[15] Vitalik Buterin. *Proposer/Builder Separation-friendly Fee Market Designs*. Ethereum Research forum post, 2021.

[16] Elijah Fox, Mallesh M. Pai, Max Resnick. *Censorship Resistance in On-Chain Auctions*. 5th Conference on Advances in Financial Technologies (AFT), LIPIcs 282, 2023.

[17] Benjamin Marsh, Alejandro Ranchal-Pedrosa, Max Resnick. *SoK: Multiple Concurrent Proposer Blockchains*. Cryptology ePrint Archive, Paper 2026/2684, 2026.
https://eprint.iacr.org/2026/2684

[18] Benjamin Marsh, Steven Landers, Jayendra Jog. *Sei Giga*. arXiv preprint arXiv:2505.14914, 2025.

[19] Kushal Babel et al. *Cadence: Extreme Pipelining with Multiple Concurrent Proposers*. 2026. arXiv:2607.02275.

[20] Giulia Scaffino, Max Resnick, Joachim Neu. *Gatling: Rapid-Fire Consensus from Parallel Composition*. 8th Conference on Advances in Financial Technologies (AFT), LIPIcs, 2026.

[21] Thomas Thiery et al. *EIP-7805: Fork-Choice Enforced Inclusion Lists (FOCIL)*. 2024.
https://eips.ethereum.org/EIPS/eip-7805

[22] Andrea Canidio. *Ordering by Unanimity: Giving Applications Sequencing Rights Without Breaking Composability*. arXiv preprint arXiv:2607.06144. 2026.