JANZEER BLOCKCHAIN

Yellow Paper

Formal protocol specification

Document v1.0 · Janzeer Network · 2026

Janzeer Yellow Paper

This document is the formal protocol specification for the Janzeer blockchain: a fixed-supply, **permissionless stake-free** Layer-1 (open validator registration + fixed-rotation block production with two-phase BFT finality) built on a custom, non-custodial HD-wallet cryptographic scheme. It defines, byte-for-byte, the cryptographic primitives, key and address derivation, transaction and block encodings, hashing, amount scaling, state-transition rules, and the consensus algorithm exactly as implemented in node version 0.1.0 (`com.zg.crypto.janzeer`). Every value here is a code default; implementers MUST reproduce these layouts exactly to remain in consensus. Where a value is a development or testnet caveat, it is flagged as such.

---

1. Notation & constants

1.1 Notation

- `||` denotes byte concatenation.
- `x.utf8` denotes the UTF-8 (ASCII for hex strings) byte encoding of string `x`. Addresses, keys, and hex strings are encoded as their **textual** bytes (the hex characters), **not** decoded to raw bytes, unless stated otherwise.
- `uint64(x)` / `int(x)` denotes a fixed-width **big-endian** integer (Java `ByteBuffer`). `(8)` = 8-byte long, `(4)` = 4-byte int.
- `H(x)` = SHA-256. `H²(x)` = `SHA-256(SHA-256(x))` (double-SHA256). `keccak(x)` = Keccak-256 (the original Keccak padding, **not** NIST SHA3-256).
- `hex(b)` = lowercase hexadecimal encoding of bytes `b`.
- `n` = the order of the secp256k1 group.
- `scaledLong(x)` = the amount-scaling function defined in §1.3.

1.2 Constants

Constant	Value	Notes
Scale (decimals)	`8`	all amounts scaled by `10^8`
`MAX_SUPPLY`	`100,000,000`	fixed; coins are never minted
`MINIMUM_AMOUNT`	`0.1`	minimum transferable amount
`MINIMUM_FEE`	`0.01`	minimum transaction fee
`MAX_DATA_SIZE`	`256` bytes	memo / data field cap
Integer endianness	big-endian	all multi-byte integers
String encoding	UTF-8 / ASCII-hex	textual bytes, not decoded
Curve	secp256k1	compressed (33-byte) public keys
HD path	`m/0/0/0`	all non-hardened
`network-id` (default)	`janzeer`	bound into every signed preimage & handshake (§1.4)
Treasury "bank" address	`0x0000000000000000000000000000000000000000`	key-less, capture-proof
`fee-create-token-tx`	`5`	exact fee to CREATE a native token (§8)
`token-activation-height`	`0`	height native tokens activate; pinned in the chain-spec digest (§8.4)

1.3 Amount scaling — `toScaledLong`

All monetary values are carried in preimages as an 8-byte big-endian signed long representing the amount times `10^8`:

```
toScaledLong(x) = x.setScale(8, HALF_UP) × 10^8   as int64 (big-endian, 8 bytes)
```

Examples: `0.01 → 1_000_000`; `0.1 → 10_000_000`; `1.5 → 150_000_000`; `10 → 1_000_000_000`.

Rounding mode is `HALF_UP` at 8 decimal places before multiplication. This function is used everywhere an amount, fee, reward, or balance enters a preimage or state-hash input.

1.4 Network identity (cross-chain replay protection)

A **network-id** string (`consensus.network-id`, default `janzeer`; distinct per chain — the mainnet, or a private test network) is bound into **both** the signed transaction preimage (§3.1) **and** the P2P handshake domain (§6.8). A transaction signed for one network therefore cannot be replayed on another: the network-id bytes are the first bytes of every transaction preimage. Every committed parity preimage now begins with `6a616e7a656572` (the UTF-8 bytes of `janzeer`).

---

2. Cryptographic primitives

2.1 Mnemonic → seed (PBKDF2)

The seed is derived from the mnemonic by PBKDF2 with HMAC-SHA512. This is **not** BIP39: the salt prefix is a Janzeer-specific literal, not the string `"mnemonic"`.

```
seed = PBKDF2-HMAC-SHA512(
          password   = NFKD(mnemonic) as UTF-8 bytes,
          salt       = "@_Janzeer_Blockchain_@" ‖ NFKD(passphrase) as UTF-8,
          iterations = 2048,
          dkLen      = 64 bytes )
```

Parameter	Value
PRF	HMAC-SHA512
Iterations	`2048`
Output length	`64` bytes
Salt prefix (literal)	`@_Janzeer_Blockchain_@`
Passphrase (default)	`""` (empty)
Normalization	NFKD on both password and passphrase

2.2 HD master key

The master node is derived by HMAC-SHA512 keyed by the **same salt string** (not BIP32's `"Bitcoin seed"`):

```
I  = HMAC-SHA512(key = "@_Janzeer_Blockchain_@", msg = seed)   // 64 bytes
IL = I[0:32]   // master private key
IR = I[32:64]  // master chain code
```

2.3 Child key derivation (non-hardened CKD)

Derivation uses the standard non-hardened CKD formula. The derivation path is `m/0/0/0` (account / wallet / address), all indices non-hardened (`i < 2^31`):

```
data       = serP(compressedParentPub)(33B) || ser32(i)(4B)
I          = HMAC-SHA512(key = parentChainCode, msg = data)
childPriv  = (I[0:32] + parentPriv) mod n
childChain = I[32:64]
```

`serP(P)` is the 33-byte compressed SEC1 encoding of the public point; `ser32(i)` is the big-endian 4-byte index. The wallet key is the private/public pair at `m/0/0/0`.

2.4 Address scheme

```
address = "0x" || hex( keccak256( compressedPubKey )[0:20] )
```

Key properties, which **differ from Ethereum**:

1. The hash input is the **33-byte compressed** public key (Ethereum uses the 64-byte uncompressed key without the `0x04` prefix).
2. The **first 20 bytes** of the digest are taken (Ethereum uses the **last 20 bytes**).
3. The hash is **Keccak-256** (original padding), not NIST SHA3-256.
4. The **canonical stored and hashed form is all-lowercase hex** with a `0x` prefix. A real **EIP-55** mixed-case checksum (`toChecksumAddress`: uppercase each hex letter whose corresponding keccak nibble is `≥ 8`) is applied at the **wallet / API display boundary only** and validated on input. It is presentational and **NOT** consensus-relevant — an all-lowercase address is always accepted, and preimages/state hashes always use the lowercase form.

2.5 Signatures (ECDSA / secp256k1)

Property	Value
Curve	secp256k1
Nonce generation	deterministic **RFC-6979** (HMAC-SHA256)
Malleability	**canonical low-S** enforced; high-S signatures are rejected
Encoding	**DER**, then **Base64**
Signed input	the **raw 32-byte digest** (no additional hashing)

The signature is computed over the 32-byte digest directly. Clients MUST sign with `prehash = false` semantics (i.e. do not let the signing library hash the input again); the digest passed in is already the final 32 bytes.

For transactions the signed digest is the **single** SHA-256 of the transaction preimage — but the transaction *identifier* (`hash`) is the double-SHA256 (§3.8). For P2P handshake proofs the signed digest is `H²(DOMAIN || ":" || networkId || nonce)` (see §6.8).

2.6 Hashing summary

Object	Hash
Transaction hash (`hash`)	`hex( H²(txPreimage) )`
Block hash	`hex( H²(blockPreimage) )`
State entry hash	`hex( H²(stateInput) )`
Merkle leaf/pair (intermediate)	`H` (single SHA-256)
Merkle final pair	`H²`, lowercase hex
Address	`keccak256(...)[0:20]`

---

3. Transaction format

Every transaction's signed preimage is the **base** layout followed by a type-specific **payload**. All integers big-endian; all addresses/keys are their UTF-8 textual bytes. The full transaction set is `TransferTx`, `PromoterTx`, `ExitPromoterTx`, and `TokenTx` (user-submitted) plus `RewardTx` and `SlashingTx` (system-generated). `TokenTx` carries a one-byte `op` selector and serves every native-token operation with a single canonical payload (§3.7, §8).

These types keep the legacy name **promoter**; a promoter is a validator.

3.1 Base preimage (every transaction)

```
networkId.utf8      (var) UTF-8 bytes of the network-id (§1.4), e.g. "janzeer"
timestamp           (8)   uint64, big-endian, epoch millis
toScaledLong(fee)   (8)   int64
nonce               (8)   uint64
senderAddress.utf8  (var) UTF-8 bytes of "0x…" address
payload()           (var) type-specific, §3.2–3.7
```

```
basePreimage = networkId.utf8 ‖ timestamp(8) ‖ toScaledLong(fee)(8) ‖ nonce(8) ‖
senderAddress.utf8 ‖ payload()
```

The `networkId.utf8` bytes are **prepended** to every preimage (this is the cross-chain replay guard of §1.4); a signature is valid only on the network whose id was signed.

3.2 TransferTx payload

```
toScaledLong(amount)   (8)   int64
recipientAddress.utf8  (var) UTF-8 bytes
(data ?? "").utf8      (var) optional memo, ≤ 256 bytes
```

Constraints: `fee ≥ 0.01`; `amount ≥ 0.1` **unless** a non-empty `data` memo is attached; `data` length ≤ `MAX_DATA_SIZE` (256).

3.3 PromoterTx payload

```
toScaledLong(amount)  (8)   int64    // the registration deposit
promoterKey.utf8      (var) UTF-8 bytes
```

Fee = **3** (exact). Deposit `amount` = **2000** (exact), transferred to the treasury bank and **non-refundable**. `promoterKey` must be unique. This registers a node as a validator (§4).

3.4 ExitPromoterTx payload

```
promoterKey.utf8      (var) UTF-8 bytes
```

Gracefully retires a validator. Fee ≥ 0.01; carries **no amount** — the registration deposit is non-refundable and is **not** returned. The validator is removed from the active set and slot schedule at the next epoch boundary (§4).

3.5 RewardTx payload (system-generated)

```
toScaledLong(reward)   (8)   int64
recipientAddress.utf8  (var) UTF-8 bytes
```

System transaction: `nonce = 0`, `fee = 0`. Never user-submitted; produced by the protocol to pay the block reward (§5.5). Its value is recomputed and validated by every node (mismatch ⇒ block rejected).

3.6 SlashingTx (system-generated)

System transaction, **fee-free**. It carries a proof of equivocation (a conflicting second vote in the same slot/stage) rather than a monetary payload; it bans the offender and pays the reporter bounty (§5.6).

3.7 TokenTx payload (native tokens, JZT-1)

`TokenTx` is the single transaction type for native tokens (§8). It begins with a one-byte `op` selector; **one canonical layout serves every op**, with each variable field length-prefixed by a 4-byte big-endian length:

```
LP(x) = int32(len(x.utf8))(4) ‖ x.utf8           // length-prefixed UTF-8

op                (1)   uint8, op ∈ {CREATE 0, MINT 1, BURN 2, SETCAP 3, TRANSFER 4}
LP(tokenId)       (var) target token id (= its CREATE tx hash); empty on CREATE
LP(symbol)        (var) 1–12 chars on CREATE; else empty
LP(name)          (var) 1–40 chars on CREATE; else empty
int32(decimals)   (4)   0–18, display-only; 0 when unused
LP(cap)           (var) max supply, decimal STRING of integer base units
LP(amount)        (var) initialSupply / mint / burn / transfer amount, decimal STRING
LP(recipient)     (var) TRANSFER destination address; else empty
```

```
tokenPayload = op(1) || LP(tokenId) || LP(symbol) || LP(name) || int32(decimals) || LP(cap)
|| LP(amount) || LP(recipient)
```

- `LP(x) = int32(len(utf8(x))) || utf8(x)`, where `int32` is a 4-byte big-endian length. **Absent fields are the empty string (length 0)** — i.e. the four length bytes `00 00 00 00` with no following bytes.
- Token `cap` and `amount` are **INTEGER base units** (arbitrary precision, like ERC-20 raw units) carried as their **canonical decimal string — NOT** `toScaledLong`. `decimals` is display-only metadata and does **not** scale the on-wire value.
- The base preimage (§3.1) is unchanged: `basePreimage = networkId.utf8 || timestamp(8) || toScaledLong(fee)(8) || nonce(8) || senderAddress.utf8 || tokenPayload`. The `fee` is paid in the **native coin** and scaled by `toScaledLong` as usual.
- Fee: CREATE pays the exact `fee-create-token-tx = 5`; MINT / BURN / SETCAP / TRANSFER pay ≥ `MINIMUM_FEE` (0.01). State-transition and authority rules are §8.

3.8 Transaction hash

```
hash = hex( H²( basePreimage ) )      // double-SHA256, lowercase hex
```

The same `basePreimage`, single-hashed, is the digest fed to the ECDSA signer (§2.5).

---

4. Block & state encoding

4.1 Block preimage (abstract)

Every block (Main or Genesis) shares the base preimage; the producer public key is part of the hashed bytes:

```
timestamp               (8)   uint64
height                  (8)   uint64
previousHash.utf8       (var) UTF-8 bytes (lowercase hex of parent)
producerPublicKey.utf8  (var) UTF-8 bytes
payload                 (var) §4.2 / §4.3
```

```
blockPreimage = timestamp(8) || height(8) || previousHash.utf8 || producerPublicKey.utf8 ||
payload
blockHash     = hex( H²(blockPreimage) )
```

Abstract block fields: `timestamp, height, previousHash, hash, signature, publicKey` plus payload.

4.2 MainBlock payload

```
txMerkle.utf8       (var) UTF-8 bytes of the tx merkle root (hex)
stateMerkle.utf8    (var) UTF-8 bytes of the state merkle root (hex)
receiptMerkle.utf8  (var) UTF-8 bytes of the receipt merkle root (hex)
```

A MainBlock carries its transactions (transfer / promoter / exit-promoter + system reward / slashing), the resulting state entries, receipts, and `epochIndex`. Capacity `block-capacity = 100` transactions.

4.3 GenesisBlock payload

```
epochIndex                  (8)   uint64
activeAnchorCount           (4)   int32, big-endian
Σ activePromoterKeys.utf8   (var) concatenated UTF-8 bytes of the frozen active-set
keys
```

The GenesisBlock persists the frozen active-validator set for the epoch **and** the number of anchors still privileged this epoch (`activeAnchorCount` — the first N of the original anchor list; see §6.2). `epochIndex` increments per genesis block. Because the privileged-anchor count is committed here, every node can reconstruct the anchor phase-out deterministically.

4.4 Merkle root algorithm

```
1. Collect leaf hashes; sort leaves in DESCENDING order.
2. Per level: pair adjacent leaves; if the count is odd, duplicate the last.
   - Concatenate the pair and hash with SHA-256 (single H) for intermediate levels.
   - For the FINAL pair, hash with double-SHA256 (H²).
3. Emit the root as lowercase hex.
```

An empty/edge input follows the same pairing rule (odd ⇒ duplicate).

4.5 AccountState hash input

```
address.utf8                (var)
scaledLong(balance)         (8)
nonce                       (8)
```

```
accountStateHash = hex( H²( address.utf8 ‖ scaledLong(balance)(8) ‖ nonce(8) ) )
```

`AccountState` carries only `balance` (spendable) and `nonce` (next expected). There is **no** `voteFor` or `accruedReward` field — voting, delegation, and reward accrual were removed in the stake-free redesign.

4.6 PromoterState

`PromoterState` carries `walletAddress` (the reward/fee destination) and `createDate` (registration time, which drives election order — §6.2), both inside the hash input:

```
promoterStateHash = hex( H²( address.utf8 ‖ walletAddress.utf8 ‖ createDate(8) ) )
```

Out-of-hash operational flags: `banned` (permanent, repeat equivocator), `exited` (gracefully retired), and the `slashedOffenses` de-duplication ledger. There is **no** `rating` field — there are no votes. Anchor status is **not** a PromoterState field: an anchor is simply a member of the epoch-1 genesis active set (§7).

---

5. State-transition functions

State is the set of `AccountState` and `PromoterState` records. A transaction applies atomically; a failed precondition rejects the transaction (and, if invalid in a block, the block). The negative-balance guard is absolute: **no balance may ever go negative**, live or during sync.

5.1 Common preconditions

- **Fee minimum/exactness**: per type (§3); `fee ≥ MINIMUM_FEE` except for `PromoterTx`, which requires the exact fixed fee `3`.
- **Nonce**: the signed preimage carries the sender `nonce`. Apply-time check is strict: `state.nonce == expected`, then `state.nonce += 1`. The mempool permits **consecutive** pending nonces (`expected = confirmedNonce + pendingCount`) for pipelining. RewardTx is **nonce-exempt**.
- **Overspend**: the sender must hold `amount + fee` (the mempool also subtracts already-pending spend before accepting a new tx).
- **Timestamp window** (protocol 3.4.0): a user transaction is valid in a block only if `block.timestamp − 6 h ≤ tx.timestamp ≤ block.timestamp + 2 min`. The comparison uses the **block's own timestamp**, never a node's wall clock, so the verdict is identical on the live path and on any later re-sync. Six hours is the mempool lifetime: `timestamp` is set when the transaction is *signed*, and a signed transaction may legitimately wait in the pool that long before a block includes it — the bound is a tolerance for slow inclusion, not a delay. Mempool intake applies the same floor and a tighter ceiling (`now + 60 s`) so that an admitted transaction can never make a later block invalid. The reward transaction is exempt (the producer stamps it with the block's timestamp). Blocks stamped before 2026-09-21T12:00:00Z predate the rule.
- **Network-id**: the signed preimage binds the network-id (§1.4); a transaction whose network-id does not match the node's is rejected.

5.2 Fee handling (all user transactions)

Fees are **split, never burned**:

```
treasuryFeeShareBps = 2000   // 20%
toBank     = fee × 2000 / 10000     // 20% → treasury bank
toProducer = fee − toBank           // 80% → block producer's wallet
```

The 20% bank share is the "D" recycling term of the emission model (§5.7).

5.3 TransferTx

- Debit sender `amount + fee`.
- Credit recipient `amount`.
- Apply fee split (§5.2).
- Increment sender nonce.

5.4 PromoterTx (validator registration)

- Require `fee = 3`, deposit `amount = 2000`, unique `promoterKey`.
- Debit sender `amount + fee`; transfer the `2000` **deposit to the treasury bank** (**non-refundable**); apply the fee split.
- Create a `PromoterState` with `walletAddress = sender`, `createDate = now`, `banned = false`, `exited = false`.
- Increment nonce.

Admission is permissionless: run a node and submit this transaction. There is no vote, no approval, and no lock-up — the non-refundable deposit plus one-seat-per-key is the Sybil guard.

5.5 ExitPromoterTx (graceful exit)

- Pay the fee (split per §5.2), increment nonce.
- Mark the validator `exited = true`; it is removed from the active set and slot schedule at the next epoch boundary.
- The registration deposit is **not** returned (non-refundable by design).

5.6 RewardTx (block reward payout)

The block reward is paid **out of the treasury bank**, not minted, to the **producing validator's wallet at 100%** — there is no delegation, commission, or reward sharing:

```
recipient = producerWallet
credit    = min( scheduledReward(h), bankBalance )   // spendable, §5.7
```

- The reward is debited from the bank and credited to the producer's spendable balance.
- The canonical value is recomputed by every node (§5.7); a mismatch rejects the block.
- RewardTx applies **first** in a block, before user transactions, and is nonce-exempt.

5.7 Emission (canonical reward value)

Emission is the fixed-supply **"C + D"** model (zero net inflation):

```
scheduledReward(h) = max( rewardTailFloor, rewardBlock / 2^( floor(h /
rewardHalvingPeriod) ) )
paidReward(h)      = min( scheduledReward(h), bankBalance )
```

Parameter	Value
`rewardBlock`	`10`
`rewardHalvingPeriod`	`2,100,000` blocks (≈ 1 year at 15-s blocks — the reward halves yearly)
`rewardTailFloor`	`0.5` (perpetual minimum)

- **C** (declining subsidy): the scheduled block reward halves every `2,100,000` blocks, floored at `0.5`, and is paid out of the pre-funded treasury bank (never minted).
- **D** (fee recycling): 20% of every fee returns to the bank (§5.2), refilling future rewards.
- When the bank empties, scheduled pay falls to 0 unless refilled by D. Net effect: no inflation, perpetual incentive.

5.8 SlashingTx (system)

- Fee-free, system-generated; carries a proof of equivocation (a conflicting second vote in the same slot/stage).
- Records an offense in the offender's `PromoterState.slashedOffenses` (de-duplicated).
- At `equivocation-ban-threshold = 2` distinct proven offenses ⇒ **permanent ban** (`banned = true`).
- Pays the reporter a bounty `slash-reporter-bounty = 5` from the treasury.

5.9 In-block ordering

Within a MainBlock, the RewardTx applies first, then user transactions in **ascending nonce** order (deterministic).

---

6. Consensus

Family: **permissionless validator block production + two-phase (PREPARE → COMMIT) BFT finality** (PBFT / Tendermint-style). Producers are **validators**; they are admitted by open registration (§5.4), not election or delegation.

6.1 Slots & epochs

Parameter	Value	Meaning
`time-slot-duration`	`12000 ms`	active production window
`time-slot-interval`	`3000 ms`	intermission
Full slot	`15000 ms` (15 s block)	duration + interval
`epoch-height`	`17000` main blocks (≈ 3 days at 15-s blocks)	epoch length

The slot owner is a deterministic **round-robin** over the epoch's frozen active-validator list. A GenesisBlock is produced at each epoch boundary and **re-elects** the active set; the epoch index increments per genesis block.

6.2 Active-set election (registration order)

Selection is by **registration order**, not rating or votes: eligible validators are ranked by `createDate` (then id) — longest-registered first.

```
activeSet = still-PRIVILEGED anchor validators (from epoch-1 genesis)
          + longest-registered elected validators
          filled up to promoters-count = 21
          − offline / banned / exited validators
```

Parameter	Value
`promoters-count`	`21` (active-set cap)
`rotating-producer-seats`	`5` (elected seats that rotate)
`producer-candidate-pool-size`	`100` (standby bench)
`equivocation-ban-threshold`	`2` offenses ⇒ permanent ban
`slash-reporter-bounty`	`5`

While registered validators ≤ the seat cap, all are active; overflow rotates fairly — the 5 rotating seats rotate round-robin through the 100-slot standby bench (registration-ordered). A validator silent on its assigned slot is ejected the next epoch (downtime). Block rewards pay 100% to the producer for the **work** of running the chain, not yield.

6.3 Automatic anchor phase-out

Anchors bootstrap the network with privilege (always-in, downtime-immune) and shed that privilege automatically, decentralizing the set over time:

Parameter	Value	Meaning
`anchor-demotions-per-epoch`	`1`	at most one anchor loses privilege per epoch
`validator-proven-epochs`	`2`	epochs a non-anchor must have produced in to count as "proven"
`safety-floor-validators`	`4`	minimum healthy set the phase-out never drops below

- An anchor is demoted (loses privilege) **one per epoch** once there are enough **proven independent** validators — non-anchors that produced in **every** one of the last `validator-proven-epochs = 2` epochs (proven/sustained, not raw registrations, so a Sybil flood cannot weaponize it).
- The phase-out never drops the healthy set below `safety-floor-validators = 4`.
- If the independent set later collapses below the floor, dormant anchors automatically **re-engage** (anti-fragile).
- Demote ≠ kill: a demoted anchor keeps validating on merit like any registrant.
- The privileged count is committed per epoch in the GenesisBlock as `activeAnchorCount` (§4.3), so it is reconstructable on every node.

6.4 Quorum

```
bftQuorum(N) = floor(2N / 3) + 1
```

over the **actual** active-set size `N`. This equals the classic `2f+1` when `N = 3f+1`, tolerating `f` Byzantine validators. Minimum launch = **4** validators (`N ≥ 3f+1`, tolerate 1 fault).

6.5 Two-phase finality

1. **PREPARE**: each active validator casts a signed PREPARE vote on a valid pending block.
2. At **quorum PREPARE** (`bftQuorum(N)` distinct PREPARE votes), a validator fully validates the block and casts a signed **COMMIT** vote.
3. At **quorum COMMIT**, the block is **final** — single-block finality, no confirmation depth.

Votes are bundled into a **QuorumCertificate** and gossiped once per slot.

6.6 Quorum certificate signing input

Each vote is an ECDSA signature (§2.5) by the validator over the block/stage being voted; votes are aggregated into the QuorumCertificate that accompanies the block. Equivocation tracking (§6.7) treats one signed vote per validator per slot per stage as canonical.

6.7 Equivocation & slashing

- A `SlotVoteTracker` enforces **one vote per validator per slot per stage**.
- A conflicting second vote (double-vote) yields a `SlashingTx` proof (§5.8).

- `equivocation-ban-threshold = 2` distinct proven offenses ⇒ permanent ban; reporter bounty `5` from treasury.

6.8 Fork choice, rollback & clock

- **Fork choice**: longest-valid-chain; the tip only moves **forward**. Single-tip — not a general reorg engine.
- **Corroborated rollback**: a rollback that would discard committed state requires the **same** `2f+1` **quorum** of distinct authenticated validators to corroborate (`RollbackCorroborator`).
- **Clock**: NTP-synced (`ntp-offset-threshold = 250 ms` — tightened for 15-s wall-clock slots). A node halts production if its clock is unsynchronized or if it is not fully synced.

6.9 P2P authentication & chain-spec pinning (consensus-relevant)

Peers complete an authenticated mutual handshake (domain `JANZEER_HANDSHAKE_V1`, folded with the network-id):

```
Greeting(nonce + chainSpecDigest) → Challenge(serverNonce + proof) → Proof(sign serverNonce)
```

Both sides verify ECDSA signatures over `H²(DOMAIN ‖ ":" ‖ networkId ‖ nonce)`; the node UID is the node public key. Only authenticated peers are admitted, and only authenticated validators' votes count toward quorum.

Chain-spec pinning: the consensus-critical parameter set (everything under `consensus.*`) is the canonical rule-set fixed at genesis. Each node computes a **chain-spec digest** (`H²` over those params) and exchanges it in the `Greeting`; a peer whose digest differs is **rejected before the challenge**. A node that edits any consensus parameter forks itself off the network instead of silently diverging. Operational settings (ports, DB, peers, NTP) stay per-node and are not part of the digest.

---

7. Genesis

7.1 Bootstrap blocks

Genesis consists of two bootstrap blocks, computed deterministically at runtime so every node derives an identical genesis:

Height	Block	Purpose
`0`	MainBlock	initial allocation (treasury + pools + anchors)
`1`	GenesisBlock	freezes the epoch-1 active set (anchor validators) + `activeAnchorCount`

7.2 Allocation

The genesis allocation sums to **exactly** `MAX_SUPPLY`, enforced by a fail-fast cap-invariant assert:

```
treasury bank (0x0000…0000)  = 77,500,000        // key-less, capture-proof
ecosystem / distribution pool = 12,000,000
founder                      =  7,500,000
three anchor validators      =  3 × 1,000,000
-------------------------------------------------
total                        = 100,000,000 = MAX_SUPPLY
circulatingSupply            = MAX_SUPPLY − bankBalance
```

The treasury bank address `0x0000000000000000000000000000000000000000` is **key-less** (capture-proof): no private key controls it; only protocol reward/fee logic moves its coins. The founder and distribution allocations (and the optional dev-fund faucet) are carved **out of the treasury** only when a custody address is supplied (`janzeer.genesis.founder` / `.distribution` / `janzeer.dev-fund`); if absent they fold back into the treasury, so nothing is minted beyond the cap.

7.3 Signer ceremony

A dedicated **ceremony key** signs **only** the two bootstrap blocks (height-0 main and height-1 genesis). It is **not** an anchor: it owns no slots and no balance. Block hash and signature are computed deterministically at runtime, so all nodes derive a byte-identical genesis. Genesis blocks are trusted-on-receipt when syncing (their active set and `activeAnchorCount` are accepted as signed, not recomputed by the receiver).

7.4 Config-driven anchors

Anchors and the ceremony are configuration-driven:

- `janzeer.genesis.anchors` — anchor validator wallets (committed dev defaults).
- `janzeer.genesis.signer-secret` — the ceremony signing key.

Caveat (development): committed anchor secrets and defaults are for development/testnet and MUST be rotated for production. The `janzeer.dev-fund` faucet (`0xaddr:amount`) is OFF by default and MUST remain off in production. Finality requires a live quorum of the active set — a **single node cannot finalize**.

---

8. Native tokens (JZT-1)

Janzeer supports optional, user-defined **fungible tokens** that live on the same chain as the native coin but form a **separate ledger**: a token's supply is **not** part of the coin's fixed `MAX_SUPPLY = 100,000,000`, and it never touches emission, the treasury, or block rewards. Every token operation still pays its **fee in the native coin**. All token operations use the single `TokenTx` type (payload §3.7) selected by a one-byte `op`.

8.1 Operations & amounts

op	Value	Authority	Fee (native coin)
CREATE	`0`	a registered validator wallet	exact `5` (`fee-create-token-tx`)
MINT	`1`	the token issuer	≥ `MINIMUM_FEE` (0.01)
BURN	`2`	the token issuer	≥ 0.01
SETCAP	`3`	the token issuer	≥ 0.01
TRANSFER	`4`	any balance holder	≥ 0.01

Token `cap`, `amount`, `totalSupply`, and balances are **integer base units** (arbitrary-precision, like ERC-20 raw units); `decimals` (0–18) is **display-only** metadata and never scales the stored or transmitted value. On the wire (§3.7) and in state-hash inputs (§8.2) these integers travel as their **canonical decimal string** — never `toScaledLong`. Persistence columns are `numeric(39,0)` (integer, no scale).

8.2 State entries

Two state records reuse the generic state / merkle / sync machinery (no new persistence engine). Each hash is `H²` (double-SHA256) of a **length-prefixed** preimage, with `LP(x) = int32(len(x.utf8))(4) ‖ x.utf8` exactly as in §3.7 (`int32` = 4-byte big-endian length).

TokenState — one per token, address `token:<tokenId>` where `tokenId` = the CREATE tx hash. Fields: `symbol`, `name`, `decimals`, `cap`, `totalSupply`, `issuer`.

```
address = "token:" ‖ tokenId
tokenStateHash = hex( H²(
    LP(address) ‖ LP(symbol) ‖ LP(name) ‖ int32(decimals) ‖ LP(cap) ‖ LP(totalSupply) ‖
LP(issuer)
) )
```

`cap` and `totalSupply` are decimal strings of integer base units; `issuer` is the creator's `0x…` address.

TokenBalanceState — one per (token, holder), address `tokbal:<tokenId>:<holder>`. Field: `balance`.

```
address = "tokbal:" ‖ tokenId ‖ ":" ‖ holder
tokenBalanceStateHash = hex( H²( LP(address) ‖ LP(balance) ) )
```

`balance` is a decimal string of integer base units.

8.3 State-transition rules (per `op`)

All rules below are enforced **deterministically and identically on the live production path AND on sync re-execution** (the transaction service's `process()`), and are additionally re-checked at mempool intake against committed state, so an un-mineable

token tx is rejected at submission (HTTP 400) rather than wasting a block slot. The invariant `0 ≤ totalSupply ≤ cap` holds after every op. The native-coin `fee` is split per §5.2 (20% → bank, 80% → producer), debited from the sender, and the sender's coin nonce is incremented per §5.1; token base units never touch the coin ledger.

CREATE (`op = 0`)

- Sender MUST be a **registered validator wallet** — a `PromoterState.walletAddress` (active or standby).
- `tokenId =` the CREATE transaction hash (globally unique; there is **no symbol registry**, so symbols are not unique).
- Set `symbol` (1–12 chars), `name` (1–40), `decimals` (0–18), `cap`, and `issuer = sender`. Create `TokenState` with `totalSupply = initialSupply`, where `0 ≤ initialSupply ≤ cap`.
- Credit `initialSupply` to the issuer's `TokenBalanceState` (`tokbal:<tokenId>:<sender>`).
- Pay the **exact** fee `5` (`fee-create-token-tx`).

MINT (`op = 1`) — sender MUST equal `issuer`. Increase `totalSupply` by `amount` and credit the issuer's balance; require `totalSupply ≤ cap` afterward. Fee ≥ 0.01.

BURN (`op = 2`) — sender MUST equal `issuer`. Decrease `totalSupply` by `amount` and debit the issuer's balance; require `totalSupply ≥ 0` afterward. Fee ≥ 0.01.

SETCAP (`op = 3`) — sender MUST equal `issuer`. Set `cap` to the new value; require `cap ≥ totalSupply` (the cap cannot drop below the minted supply). Fee ≥ 0.01.

TRANSFER (`op = 4`) — **any holder** may move `amount` base units of `tokenId` from its own `TokenBalanceState` to the `recipient`'s. Debit the sender's balance and credit the recipient's (creating the recipient's balance row if absent); the sender's token balance may not go negative. `totalSupply` is unchanged. Fee ≥ 0.01.

8.4 Activation & protocol version

- The activation gate **`token-activation-height`** (default `0` — live from genesis on a fresh network) is part of the consensus parameter set and is therefore **pinned in the chain-spec digest** (§6.9). A network thus agrees on the exact height at which token operations turn on, and a node that changes it forks itself off.
- Native-token support bumped the application **`PROTOCOL_VERSION`** **to** **`1.1.0`** (the REST envelope `version` field) and the **P2P wire** **`protocol-version`** **to** **`3.0.0`** (a new `MessageType` and typed block lists), so a pre-token node cleanly refuses to peer with a token-aware node (§6.9).

---

9. Conformance / parity vectors

Byte-for-byte conformance is proven by committed parity vectors that cover HD derivation, address generation, transaction hashing, and signature production for transfer, promoter, and exit-promoter transactions, plus native-token CREATE and TRANSFER (`tokenCreateTx`, `tokenTransferTx`):

- `build/wallet-parity-vectors.json`
- each wallet's `test/` (frontend mirrors)

Canonical test vector:

```
mnemonic : abandon abandon abandon … abandon about   (the BIP39 "abandon…about"
mnemonic)
address  : 0x06e1c0fa9955a700876f8cb0acc7f13fba9fb8ba
```

Because the network-id is now prepended to every preimage, each committed transaction preimage **begins with** `6a616e7a656572` (the UTF-8 bytes of `janzeer`).

An independent implementation conforms if it reproduces this address from the canonical mnemonic (via §2.1–§2.4) **and** reproduces the committed transaction hashes and signatures (§2.5, §3). Note that, because signatures are deterministic (RFC-6979, low-S, DER, Base64) over the single-SHA256 digest, a conforming signer yields **identical** signature bytes for identical inputs.

Document status: version 1.0. Reflects node version 0.1.0 (`com.zg.crypto.janzeer`), including native tokens (JZT-1, §8) — application `PROTOCOL_VERSION` `1.1.0`, P2P wire `protocol-version` `3.0.0`. This Yellow Paper is the authoritative source for exact byte-level facts (preimages, hashing, scaling, derivation); sibling documents (White Paper, Developer Guide, User Guide, Operator Guide, Glossary) defer to it for those details.