Rede Server Technische Dokumentation

VERSION: V2.19.11-BETA | PROTOKOLLVERSION: 3 | LIZENZ: AGPL-3.0

GEARFISH TEAM: KEANU AMANN | DAVID KLEINFERCHER

Dieses Dokument verschriftlicht ausschließlich den **Server**. Die Client-Seite (Avalonia v2 und Terminal v1) wird im Rede-Client Dokument behandelt. Das Wire-Protokoll wird in beiden Dokumenten beschrieben - hier aus der Sicht dessen, was der Server empfängt, validiert und weiterleitet.

Inhalt

Rede Server Technische Dokumentation ... 1

Version: v2.19.11-beta | Protokollversion: 3 | Lizenz: AGPL-3.0 ... 1

1. Projektstruktur ... 3

2. Architekturüberblick (Server-Sicht) ... 4

3. Wire-Protokoll und Strukturvalidierung ... 5

3.1 Nachrichtenformat ... 5

3.2 Nachrichtentypen (übersichtsweise) ... 6

3.3 Strukturvalidierung in `handleMessage` ... 7

3.4 Authentifizierung (Client-Server-Tanz) ... 9

3.5 Nachrichtenumschlag (Envelope) - was im `encrypted`-Feld steckt ... 10

4. Server-Architektur ... 11

4.1 Dateistruktur ... 11

4.2 Datenbankschicht ... 11

4.3 Clustering, Redis und Föderation ... 12

4.4 SFU (mediasoup) ... 18

4.5 Blob-Speicher (Anhänge) ... 18

4.6 Ratenbegrenzung ... 19

4.7 Verbindungswarteschlange ... 20

4.8 Verbindungs-Lifecycle und Message-Dispatch ... 21

4.9 Auth-Flow und Multi-Device-Registrierung ... 23

4.10 Pre-Keys, OTP-Reservierung und Sealed Sender (Server-Pfad) ... 25

4.11 Voice-Call-Server (1:1 SRTP + Gruppen-LiveKit) ... 28

4.12 Server-Signaturen, Nonce-Replay, Strukturvalidierung ... 29

4.13 Lifecycle, Cleanups, Invites, Status ... 31

5. DB-Schema (was der Server speichert) ... 33

6. Sicherheitsmodell (Server-Sicht) ... 35

7. Deployment ... 37

7.1 Einzelknoten ... 37

7.2 Föderiertes Deployment (Multi-Host-Cluster) ... 38

7.3 Patch-Workflow auf zwei Nodes ... 41

8. Abhängigkeiten ... 43

Server (Node.js) ... 43

Infrastruktur (Föderation) ... 43

1. Projektstruktur

Der Server lebt unter `/home/amke/Rede/server/` (kanonischer Code auf node-a). Dies ist ein **privates** Git-Repository, nicht öffentlich. Eine Mirror-Kopie liegt unter `/home/amke/Rede/rede-server-repo/` und wird zum privaten GitHub-Repo `caaatto/rede-server` gepusht - ausschließlich zum Verteilen an förderierte Nodes.

```
/home/amke/Rede/server/
|-- index.js                  # Hauptserver, alle Handler (3159 Zeilen)
|-- store.js                  # SQLite-Speicher (1236 Zeilen)
|-- pg-store.js               # PostgreSQL-Speicher (1149 Zeilen)
|-- redis.js                  # Redis Shared State (561 Zeilen)
|-- cluster.js                # Node.js Clustering (206 Zeilen)
|-- sfu.js                    # mediasoup SFU (437 Zeilen)
|-- loadtest.js               # WebSocket-Lasttests (173 Zeilen)
|-- migrate-sqlite-to-pg.js   # Einmalige SQLite -> PostgreSQL Migration
|-- invite.js                 # Invite-Code-Logik
|-- rede-server.service       # systemd-Unit
|-- livekit/                  # LiveKit JWT-Minting für Gruppenanrufe
+-- .env                      # Passphrase + Connection-URLs (NICHT in Git)
```

Codeumfang: ~6.800 Zeilen JavaScript.

Das `shared/protocol.js` (147 Zeilen, Protokoll-Konstanten) existiert in **zwei** Kopien: `/home/amke/Rede/shared/protocol.js` für den Server und `/home/amke/Rede/rede-client/shared/protocol.js` für den Client. Beide müssen byte-identisch bleiben.

2. Architekturüberblick (Server-Sicht)

Der Server ist ein **blindes Relay**. Er sieht ausschließlich verschlüsselte Daten. Klartext existiert nur auf den Endgeräten.

```
+-------------------+          +--------------------+          +-------------------+
|   Client A        |          |    Server          |          |   Client B        |
|                   |   E2EE   | [Relay + Storage]  |   E2EE   |                   |
|  Klartext         +--------->   WebSocket         +--------->  Klartext          |
|  X3DH/PQXDH       |          |   SQLite/PG        |          |  X3DH/PQXDH       |
|  Double Ratchet   |          |   Redis Pub/Sub    |          |  Double Ratchet   |
|  Sender Keys      |          |   LiveKit-JWT      |          |  Sender Keys      |
|  SRTP/SFrame      |          |                    |          |  SRTP/SFrame      |
|                   |  <-----  |                    |  <-----  |                   |
+-------------------+  Binary  +-------------------+   Binary  +-------------------+
                       Frames                          Frames
```

Was der Server tut:

- Nimmt WebSocket-Verbindungen entgegen (TLS-Terminierung typischerweise via nginx davor).
- Authentifiziert mit Challenge-Response (Ed25519).
- Validiert Nachrichten **strukturell**, ohne ihren Inhalt zu lesen.
- Persistiert Pre-Key-Bundles, Mitgliederlisten und Pending-Messages (alle Felder field-level verschlüsselt at rest).
- Routet Nachrichten zwischen Clients und - in der Föderation - zwischen Nodes via Redis Pub/Sub.
- Signiert alle ausgehenden Nachrichten mit Ed25519 (TOFU-gepinnt beim Client).

Was der Server nicht tut:

- Klartextnachrichten lesen.
- Den Absender einer Sealed-Sender-Nachricht ermitteln.
- Media-Inhalte von Anrufen entschlüsseln (binäre SRTP-Frames werden nur weitergeleitet).
- Den sozialen Graphen (Kontaktlisten) ableiten - Statusbroadcasts gehen an alle Online-User.

3. Wire-Protokoll und Strukturvalidierung

Diese Section beschreibt das Format aller Nachrichten zwischen Client und Server. Was der Client damit lokal anstellt, steht in der Client-Doku.

3.1 NACHRICHTENFORMAT

Alle Nachrichten sind JSON-kodiert und werden über WebSocket transportiert. Eingang ist `wss://<host>/rede`.

Server -> Client:

```
{
  "v": 3,
  "type": "message_type",
  "ts": 1712505600000,
  "serverSig": "Base64(Ed25519-Signatur)",
  ...payload
}
```

Client -> Server:

```
{
  "v": 3,
  "type": "message_type",
  ...payload
}
```

Timestamps werden vom Server gesetzt, nicht vom Client. Jede vom Server ausgehende Nachricht trägt eine Ed25519-Signatur (`serverSig`) - siehe §4.12. `v` muss `3` sein, sonst Reject mit generischem ERROR.

3.2 NACHRICHTENTYPEN (ÜBERSICHTSWEISE)

Kategorie	Typen	Richtung
Authentifizierung	`register`, `auth`, `auth_challenge`, `auth_response`, `auth_ok`, `auth_fail`	Bidirektional
Pre-Keys (X3DH/PQXDH	`upload_prekeys`, `fetch_prekey_bundle`, `prekey_bundle`	Bidirektional
1:1-Nachrichten	`message`, `message_ack`, `sealed_message`, `sealed_message_ack`	Bidirektional
Gruppen	`group_create`, `group_invite`, `group_kick`, `group_message`	Bidirektional
Places	`place_create`, `place_invite`, `place_kick`, `place_leave`, `place_channel_add/remove`, `place_message`, `place_role_set`, `place_ban/unban`, `place_members`	Bidirektional
Sprachanrufe	`call_offer`, `call_answer`, `call_ice`, `call_hangup`, `call_reject`, `call_busy`, `call_ringing`	Bidirektional
Gruppenanrufe	`gcall_reqüst_token`, `gcall_token`, `gcall_announce`, `gcall_end`, `gcall_active`	Bidirektional
Blobs	`blob_upload`, `blob_upload_ok`, `blob_fetch`, `blob_data`	Bidirektional
Status	`status_update`, `status_change`	Bidirektional
Geräte	`device_link_create`, `device_link_use`, `device_added`	Bidirektional
Einladungscodes	`invite_create`, `invite_create_ok`	Client -> Server (admin-Key gegated)
Warteschlange	`qüü_position`, `qüü_admit`	Server -> Client
System	`error`, `pending_messages`, `session_end`	Server -> Client

3.3 STRUKTURVALIDIERUNG IN `HANDLEMESSAGE`

Lauft VOR jeder Krypto-Operation, gegen Heap-Erschöpfung und Format-Probing:

Feld	Prüfung
`encrypted`	base64-decodierbar, mindestens 16 Zeichen
`nonce`	base64, decodiert exakt 24 Byte
`header.dh`	base64 32 Byte (Curve25519-Pubkey)
`header.n`	Integer, >= 0, <= 100 000
`header.pn`	Integer, >= 0, <= 100 000
`header` gesamt	`JSON.stringify(header).length <= MAX_HEADER_SIZE` (= 1024)
`x3dh.identityKey`, `.ephemeralKey`, `.usedOTPKPub`	base64 32 Byte
`x3dh` gesamt	`JSON.stringify(x3dh).length <= MAX_X3DH_SIZE` (= 4096)
`senderKeyHeader.messageNumber`	Integer, >= 0, <= 10 000
`senderKeyHeader.signature`	base64 64 Byte (Ed25519)
`senderKeyHeader` gesamt	`<= MAX_HEADER_SIZE` (= 1024)
`sealedPayload.ciphertext`	base64, `<= 32768` Zeichen
`ttl`	`sanitizeTTL` - non-negative safe Integer (verhindert `NaN`-Memory-Leak in Per-Message-Timern, Audit-Fix H3)

Warum zwei Header-Caps statt einem (2026-05-26-Regression): Vor PQXDH teilten sich Double-Ratchet-Header und x3dh-Block einen gemeinsamen `MAX_HEADER_SIZE = 1024`. Mit PQXDH (v2.19.0) bringt der x3dh-Block den ML-KEM-768-Ciphertext mit (1088 Byte roh, ~1452 base64 chars) plus identityKey, ephemeralKey, usedOTPKPub und usedPQOTPKPub - in Summe ~1700 chars JSON. Das alte 1024-Limit hat jede frische Session mit `Invalid or oversized x3dh data` abgelehnt, ohne dass irgendein Client-Service einen Handler für `MSG.ERROR` registriert hatte - die Fehler verschwanden lautlos. Seit dem Fix gilt `MAX_X3DH_SIZE = 4096` separat für den x3dh-Block (Headroom für künftige PQ-Felder); `MAX_HEADER_SIZE` bleibt 1024 für den reinen DR-Header.

Sealed-Sender-Cap (2026-05-26-Regression): `handleSealedMessage` lehnte Ciphertexte > 24000 chars ab. Ein Sealed-Envelope umhüllt eine vollständig gepaddete DR-Nachricht - grösster Bucket 16382 -> +16 secretbox -> base64 ~21864 -> innere JSON ~22100 -> +16 nacl.box -> base64 ~29500. Profile-Sync-Nachrichten mit Avatar landen genau in diesem Bereich und wurden durch die Bank rejected. Cap ist jetzt 32768. Beim Anpassen der Padding-Buckets immer auch hier nachprüfen.

WebSocket-Frame-Limit: `MAX_PAYLOAD = 32 KB` (in `ws`-Library konfiguriert). Daraus folgend bricht der Server eine Verbindung mit Code `1009 "Message too large"` ab, sobald ein Frame das überschreitet - diese Schicht greift VOR der Strukturvalidierung.

3.4 AUTHENTIFIZIERUNG (CLIENT-SERVER-TANZ)

Domain Separation: Die Challenge wird als `"AUTH_CHALLENGE:" + challenge` signiert (nicht die rohen Challenge-Bytes). Server und Client müssen dieselbe Präfixierung verwenden; sonst scheitert die Verifikation.

Anti-Enumeration: Existiert der angefragte User nicht, sendet der Server trotzdem eine *fake* Challenge gleicher Form und gleicher RTT, markiert sie intern als `fake=trü`. Erst nach `AUTH_RESPONSE` antwortet er mit generischem `AUTH_FAIL`. So leakt der Server nicht über Timing oder unterschiedliche Antwortpfade, ob eine UserID existiert.

3.5 NACHRICHTENUMSCHLAG (ENVELOPE) - WAS IM `ENCRYPTED`-FELD STECKT

Was der Client innerhalb des ratchet-verschlüsselten Klartext-Blobs sendet, ist ein einfaches JSON-Envelope:

```
{
  "t": "Nachrichtentext",
  "ref": "msgId (Reply)",
  "rp": "Reply-Vorschau (max 100 Zeichen)",
  "ra": "Reply-Autor",
  "att": [{ "bid": "blobId", "key": "...", "nonce": "...", "name": "...", "mime": "...", "size": 0 }]
}
```

Einfache Strings ohne JSON-Struktur werden vom Client als Legacy-Nachrichten interpretiert (Rückwärtskompatibilität). Der Server sieht weder den Klartext noch dieses Envelope - er routet nur den `encrypted`-Blob.

Kontrollnachrichten (`__rede_ctrl: "profile" | "reaction" | "edit" | "delete"`) gehen über **dieselbe** verschlüsselte Schicht. Der Server unterscheidet sie nicht von normalen Texten.

4. Server-Architektur

4.1 DATEISTRUKTUR

Datei	Zeilen	Funktion
`index.js`	3159	WebSocket-Server, alle Nachrichtenhandler, Ratenbegrenzung
`store.js`	1236	SQLite-Backend (better-sqlite3, WAL-Modus)
`pg-store.js`	1149	PostgreSQL-Backend (async, Connection Pool)
`redis.js`	561	Redis Shared State (Sessions, Rate Limits, Pub/Sub)
`cluster.js`	206	Multi-Worker-Clustering, IPC-Passphrase-Verteilung
`sfu.js`	437	mediasoup SFU (Opus, WebRtcTransport, SRTP)
`loadtest.js`	173	WebSocket-Kapazitätsbenchmark
`migrate-sqlite-to-pg.js`	293	Einmalige SQLite -> PostgreSQL Migration
`invite.js`	—	Invite-Code-Erzeugung / -Validierung
`livekit/`	—	JWT-Minting + Scope-Routing für Gruppenanrufe

4.2 DATENBANKSCHICHT

SQLite (store.js):

- `better-sqlite3` mit WAL-Modus für parallele Lesezugriffe
- Tabellen: `users`, `prekey_bundles`, `pending_messages`, `groups`, `group_members`, `places`, `place_members`, `place_admins`, `place_bans`, `invite_codes`, `device_links`, `blobs`, `nonces`, `_meta`
- Feldbezogene `nacl.secretbox`-Verschlüsselung mit dem Master-Key (scrypt aus Server-Passphrase)
- **Synchrone API** - keine `await` nötig

PostgreSQL (pg-store.js):

- Drop-in-Ersatz für `store.js`
- **Asynchrone API** - alle Funktionen geben Promises zurück, Aufrufer müssen `await` verwenden
- Connection Pool (`pg`-Modul)
- Identische Verschlüsselung wie SQLite
- Voraussetzung für den Föderations-Modus, weil mehrere Nodes auf denselben Datensatz schreiben

Umgebungsvariable: `REDE_DB_BACKEND=pg|sqlite` wählt das Backend.

Achtung beim Editieren von `index.js`: Beide Backends müssen synchron gehalten werden. Wenn `pg-store.js` eine neü Funktion bekommt, muss sie auch in `store.js` existieren (mit synchroner Semantik). Die Migration `migrate-sqlite-to-pg.js` deckt nur einen One-Shot-Import ab; im laufenden Betrieb laufen die Backends nie nebeneinander.

4.3 CLUSTERING, REDIS UND FÖDERATION

Der Server skaliert in drei Stufen, alle aus demselben Code (`server/index.js`):

1. **Einzelprozess** (`node server/index.js`) - kompletter Zustand in In-Memory-Maps. Standardmodus, wenn `REDE_REDIS_URL` nicht gesetzt ist.
2. **Cluster** (`node server/cluster.js`) - Master forkt N Worker auf *einer* Maschine, alle teilen sich den Listening-Socket. Worker-zu-Worker-Routing läuft über Redis.
3. **Föderation (Multi-Host-HA-Cluster)** - mehrere Server auf verschiedenen Maschinen teilen sich *eine* PostgreSQL und *eine* Redis. Redis Pub/Sub routet Nachrichten zwischen den Hosts. Reverse-Proxy (nginx) davor verteilt Clients per Round-Robin und macht Failover passiv.

Die Auswahl steürn Umgebungsvariablen. Ohne `REDE_REDIS_URL` bleibt das Verhalten bit-identisch zum klassischen Einzelprozess-Modus.

Redis-Schema

`server/redis.js` definiert das gemeinsame Layout:

Schlüssel	Inhalt	TTL
`clients:{userId}`	Hash `deviceId -> {workerId, wsId}` (globale Client-Directory)	bis zur Trennung
`worker:{workerId}`	Pub/Sub-Kanal für Cross-Worker-Routing	--
`worker-alive:{workerId}`	Liveness-Heartbeat, alle 10 s erneürt	30 s
`activecall:{callId}`	JSON `{callerId, calleeId, mode, startedAt}` (1:1-Anrufe)	4 h
`rl:{type}:{key}`	Counter für verteilte Rate Limits	window
`ipconn:{ipHash}`	Counter für Verbindungen pro IP	--
`challenge:{wsId}`	Auth-Challenge (optional, Fallback)	60 s
`deliveryTokenSecret`	HMAC-Schlüssel für Sealed-Sender-Tokens	24 h

Cross-Node-Nachrichten-Routing

`redis.sendToUser(uid, msg)` (`message` ist String oder `Buffer`):

1. `getClientDevices(uid)` liest den Redis-Hash `clients:{uid}`.
2. Für jeden Eintrag: ist `workerId === eigener Worker`, geht direkt `localWsMap.get(wsId).send(msg)`.
3. Sonst `publishToWorker(workerId, { action: 'send', wsId, message, binary })` auf den Kanal `worker:{workerId}` des Ziel-Workers.
4. Der Ziel-Worker hat `redis.onCrossWorkerMessage(handler)` registriert; der Handler ruft `getLocalWs(wsId).send(decoded)` auf.

Binäre Payloads (SRTP-Frames in 1:1-Sprachanrufen) werden Base64-kodiert und mit `binary: trü` markiert; der Empfänger dekodiert sie zu einem Buffer. Voice-Anrufe funktionieren damit auch über Maschinengrenzen.

Die Nachricht ist auf der **Ursprungs-Node** bereits durch `signedMessage(...)` Ed25519-signiert; die Empfänger-Node leitet sie nur weiter und signiert nicht erneut - das Sicherheitsmodell bleibt unverändert.

Worker-Liveness und Stale-Cleanup

Beim Start ruft jeder Server `redis.startHeartbeat()` auf. Stirbt eine Node, läuft `worker-alive:{workerId}` nach 30 s ab. `getClientDevices` prüft für jeden nicht-lokalen Eintrag, ob das Heartbeat-Key noch existiert; tut es das nicht, wird der Hash-Eintrag mit `HDEL` aufgeräumt - die Client-Directory heilt sich selbst, ohne separaten Cleanup-Job. Beim regulären Shutdown löscht `redis.shutdown()` den eigenen Heartbeat sofort, damit Peers nicht 30 s lang noch dorthin routen.

1:1-Call-Registry

Damit der binäre SRTP-Relay-Check auch auf der Callee-Node passt, wird `activeCalls` beim `CALL_OFFER` zusätzlich in Redis abgelegt (`activecall:{callId}`, 4-h-TTL). Bei `CALL_ANSWER` auf der Callee-Node liest deren Handler den Eintrag aus Redis und hydriert die lokale Map. So kann der Binary-Relay-Pfad auf beiden Seiten den Peer bestimmen, selbst wenn `OFFER` und `ANSWER` auf verschiedenen Maschinen verarbeitet wurden.

pg-store: gemeinsame Server-Identität via `_meta`

Statt `master_salt`, `server_signing_key` und `admin_key` als lokale Dateien zu halten, legt `pg-store.js` sie in der Tabelle `_meta` ab. Der Bootstrap läuft unter `pg_advisory_lock(1380863045)`, sodass zwei parallel startende Nodes nicht racen:

```
schema (CREATE TABLE IF NOT EXISTS)
  -> master_salt aus _meta lesen (oder generieren / aus data/-Datei importieren)
  -> Master-Key ableiten (scrypt N=2^20)
  -> passphrase_check verifizieren
  -> ggf. Legacy-Scrypt-Upgrade
  -> server_signing_key + admin_key laden (oder importieren / generieren)
```

- **Frische Node:** Mit nur `REDE_PG_URL` und der korrekten Passphrase zieht sie beim ersten Start automatisch die Server-Identität aus `_meta`.
- **Pre-existierende Single-Node:** Ihre `data/.master_salt`, `data/.server_signing_key` und `data/.admin_key` werden beim ersten pg-Start einmalig in `_meta` importiert; die TOFU-Identität (Ed25519-Server-Signing-Key) bleibt über den Umstieg erhalten.

Was bleibt pro Node lokal

Bewusst nicht über Redis verteilt (Trade-off zwischen Performance und Korrektheit):

- `pendingChallenges` - Auth-Flow ist an genau eine WS-Verbindung gebunden, daher per Definition node-lokal.
- `connectionQüü` und `MAX_CONNECTIONS` - Socket-Kapazität pro Node; Round-Robin im Reverse-Proxy verteilt neü Verbindungen ohnehin.
- `userStatuses` und `broadcastStatusChange` - Online/Offline-Statuses werden nur an lokal verbundene User gesendet (per-Node best-effort).
- ~20 In-Memory-Rate-Limit-Maps - per-Node; mit Round-Robin-LB kann ein Nutzer ein Limit effektiv um den Faktor N (Anzahl Nodes) erhöhen. Akzeptiert für die Lastverteilung; sicherheitskritische Limits sind optional über `redis.checkRateLimit('targetauth', ...)` global ziehbar.

Umgebungsvariablen

Variable	Wirkung
`REDE_DB_BACKEND`	`sqlite` (Default, Einzelprozess) oder `pg` (geteilte DB, Voraussetzung für Föderation)
`REDE_PG_URL`	PostgreSQL-Verbindungsstring (Pflicht im pg-Modus)
`REDE_REDIS_URL`	Redis-URL; nicht gesetzt -> In-Memory-Fallback (kein Cluster, keine Föderation)
`REDE_NODE_ID`	Stabile Worker-ID im Einzelprozess-Modus (z.B. `node-a`, `node-b`). Im Cluster-Modus hängt `cluster.js` zusätzlich `-w0`, `-w1`, ... an
`REDE_WORKER_ID`	Wird von `cluster.js` pro Worker gesetzt - hat Vorrang vor `REDE_NODE_ID`
`REDE_BIND`	Bind-Adresse des WS-Servers (Default `127.0.0.1`; auf einer entfernten Node z.B. deren Tunnel-IP)
`REDE_TRUSTED_PROXIE`	Komma-getrennte IPs, die wie `localhost` behandelt werden (Reverse-Proxies, vom Per-IP-Connection-Cap ausgenommen)
`REDE_NO_I2P`	Wenn gesetzt, kein I2P-Listener (sinnvoll auf Nodes, die nur über den Reve Proxy erreichbar sein sollen)
`REDE_DEBUG_USER`	Komma-Liste von UserIDs, für die `dlog(...)` aktiv ist (verbose journal-Logs). Andere User werden NICHT geloggt. Leer / unset = kein Debug-Output.

Betriebsmodi (Zusammenfassung)

- `npm run server`: Einzelprozess, SQLite, kein Redis. Einfachster Standalone-Betrieb.
- `npm run cluster`: Multi-Worker auf *einer* Maschine; Redis empfohlen für Cross-Worker-Routing innerhalb des Hosts.
- `systemctl start rede-server` mit `REDE_DB_BACKEND=pg` + `REDE_REDIS_URL`: Föderation. Eine Node oder N Nodes - beliebig erweiterbar.

Bekannte Föderations-Edge-Cases

Zwei theoretische Silent-Drop-Pfade im Cross-Node-Routing sind dokumentiert, aber bislang nicht beobachtet als echte Ursache von Nutzer-gemeldeten Symptomen:

1. **\`publishToWorker\` ist fire-and-forget.** `redis.publish` liefert nur an *aktüll subscribierte* Worker - ist der Ziel-Worker zwischen Disconnect und neüm `subscribe` (z.B. während eines `ws.close 1006`-Reconnect-Bursts), geht die Nachricht ins Leere, aber `sendToDevice`/`sendToUser` returnen trotzdem `trü`. Der Sender bekommt `MESSAGE_ACK`, die Nachricht landet *nicht* in `pending_messages`. Realer Fix wäre: `redis.publish`-Returnwert (Subscriber-Count) auswerten und bei 0 auf `addPendingMessage` umlenken.
2. **\`getPendingMessages\` löscht vor Auslieferungsbestätigung.** Die Pending-Auslieferung in `deliverPending` macht in *einer* Transaktion `SELECT ... DELETE ...` und sendet dann `PENDING_MESSAGES` per `ws.send` ohne Client-ACK. Bricht der Socket während des Sends ab (1006), sind die Pendings weg. Realer Fix: ACK-Roundtrip ins Protokoll aufnehmen, `DELETE` erst nach Bestätigung.

In der Praxis hat nginx-LB die Clients meist auf denselben Worker geroutet, und die in §4.12 / §3.3 dokumentierten Wire-Size-Caps haben die Symptome dominiert - die hier genannten Pfade sind als "warten auf konkreten Wiederholungsfall" markiert, nicht als unmittelbare Priorität. Zum Live-Debug existiert `/tmp/rede-pubsub-tap.js`: ein Read-Only-Helfer, der per Pattern-Subscribe alle `worker:*`-Kanäle mitliest und Aktion/`wsId`/Message-Type loggt.

4.4 SFU (MEDIASOUP)

```
Datei: server/sfu.js (437 Zeilen)
```

mediasoup-basierte Selective Forwarding Unit für Gruppenaudio:

- Nur Opus-Codec
- WebRtcTransport + PlainTransport mit SRTP
- Raum-Management: max 50 Räume, 25 Teilnehmer, 4 h Timeout
- Noch nicht vollständig in `index.js` integriert; 1:1 Anrufe nutzen direktes Binary-Frame-Relay (§4.11), Gruppenanrufe laufen über externes LiveKit + SFrame (siehe Client-Doku).

4.5 BLOB-SPEICHER (ANHÄNGE)

Dateianhange werden als opake Blobs gespeichert:

- Max. 10 MB pro Blob
- 30 Tage TTL (stündlicher Bereinigungsjob)
- Rate Limit: 20 Uploads pro Stunde pro Benutzer
- SQLite-Tabelle `blobs` (blobId, data als Base64, size, uploaded_at, user_hash)
- Server sieht nie Dateinamen, MIME-Typ oder Inhalt (clientseitig nacl.secretbox-verschlüsselt)

Der Client cacht den Chiffretext lokal - so bleibt der Inhalt für den User auch nach der 30-Tage-TTL des Servers verfügbar. Details dazu in der Client-Doku.

4.6 RATENBEGRENZUNG

Alle Rate-Limit-Maps verwenden gehashte Schlüssel (`rlHash = SHA256(...).slice(0,16)`), sodass User-IDs nicht im Klartext im Speicher liegen. Die Maps sind per-Node-lokal - in einem Föderations-Cluster mit `N` Nodes erhöhen sich die effektiven Limits um Faktor `N` (akzeptiert, s. §4.3). Ausnahme: `targetAuthLimits` (Anti-Enumeration) kann optional global via `redis.checkRateLimit('targetauth', ...)` gezogen werden.

Map	**Schlüssel**	**Limit**	**Fenster**	**Wo**
`rateLimits`	userId	30	1 min	jeder Dispatch eines authentifizierten Users
`authRateLimits`	ipHash	5	1 min	unauth. `AUTH`/`REGISTER`/`AUTH_RESPONSE`/ `INVITE_CREATE`
`targetAuthLimits`	targetUserId	10	1 min	`AUTH` (gegen User-Enumeration)
`inviteRateLimits`	adminKeyHash	10	1 h	`INVITE_CREATE`
`prekeyFetchLimits`	senderId	20	1 min	`FETCH_PREKEY_BUNDLE`
`otpConsumptionLimits`	targetId	20	1 h	`FETCH_PREKEY_BUNDLE` (OTP-Erschöpfung verhindern)
`lookupLimits`	senderId	20	1 min	`USER_LOOKUP`
`deviceLinkUseLimits`	targetUserId	5	1 min	`DEVICE_LINK_USE`
`groupCreateLimits`	senderId	5	1 h	`GROUP_CREATE`
`groupInviteLimits`	senderId	10	1 min	`GROUP_INVITE`
`groupKickLimits`	senderId	10	1 min	`GROUP_KICK`
`groupMsgLimits`	senderId	30	1 min	`GROUP_MESSAGE`
`placeCreateLimits`	senderId	5	1 h	`PLACE_CREATE`
`placeInviteLimits`	senderId	10	1 min	`PLACE_INVITE`
`placeKickLimits`	senderId	10	1 min	`PLACE_KICK`
`placeMsgLimits`	senderId	30	1 min	`PLACE_MESSAGE`
`blobFetchLimits`	senderId	20	1 min	`BLOB_FETCH`

Map	Schlüssel	Limit	Fenster	Wo
`sealedRateLimits`	ipHash	30	1 min	`SEALED_MESSAGE` (kein userId verfügbar)
`callSignalingLimits`	senderId	20	1 min	`CALL_*`-Signaling + SFU-Control
`gcallTokenLimits`	senderId	10	1 min	`GCALL_REQÜST_TOKEN`

Zusätzlich `otpReservations` - keine Begrenzung, sondern eine Map `rlHash(senderId:targetId:deviceId) -> {otpkId, expiresAt}` mit 5-min-TTL, um wiederholte Bundle-Fetches idempotent zu halten und Eager-OTP-Verbrauch zu vermeiden (Details in §4.10).

4.7 VERBINDUNGSWARTESCHLANGE

```
MAX_CONNECTIONS: 4000 (getestet auf 8-Core EPYC, 16GB RAM, 712MB RSS)
```

Bei voller Kapazität:

1. Neü Verbindungen werden in eine Warteschlange eingereiht (max 500, 5 Minuten Timeout)
2. `QÜÜ_POSITION`-Updates alle 3 Sekunden
3. `QÜÜ_ADMIT` wenn ein Platz frei wird
4. `drainQüü()` bei jedem Disconnect + periodisch
5. Client wartet auf `QÜÜ_ADMIT` bevor Auth gestartet wird

4.8 VERBINDUNGS-LIFECYCLE UND MESSAGE-DISPATCH

```
Datei: server/index.js (~3160 Zeilen - alle Handler in einer Datei)
```

Eine Client-WS-Verbindung durchläuft:

```
wss.on('connection') - IP-Counting - Qüü oder setupConnection()
                                                    |
                                ws.on('message', isBinary)
                                                    |
                  +----- isBinary -------------+----- text -------+
                  |                                               |
              C3-SRTP-Relay                  parseMessage + Version-Check
                  |                          + Rate-Limit-Hook + switch(msg.type)
              sendToUser(peer)                       |
                                              handle<X>(ws, msg, userId, deviceId)
                                                     |
                                              redis.registerClient (REGISTER/AUTH/LINK)
                                                     |
                                                    ...


wss.on('close')
  await redis.unregisterClient(userId, deviceId)
  redis.unregisterWs(ws)
  broadcastStatusChange(uid, 'offline') wenn isUserOnline=false (cross-node!)
  drainQüü()
```

1. `wss.on('connection')`:

- Liest `req.socket.remoteAddress`. Localhost (`127.0.0.1`, `::1`, `::ffff:127.0.0.1`) und Adressen aus `REDE_TRUSTED_PROXIES` bekommen pro Verbindung eine zufällige `connId` (zwei verschiedene Clients hinter demselben Proxy teilen sich keine Per-IP-Limits). Andere Verbindungen verwenden `remoteAddr` direkt.
- `ipHash = SHA256(connId).slice(0, 16)` - Klartext-IPs landen nicht in den Map-Schlüsseln.
- Per-IP-Connection-Cap: `MAX_CONNECTIONS_PER_IP = 5` (übersprungen für Localhost + Trusted Proxies).
- Bei `activeConnectionCount >= MAX_CONNECTIONS (4000)`: in `connectionQüü` einreihen (s. §4.7), sonst direkt `setupConnection(ws, ipHash, isLocalhost)`.

2. `setupConnection`:

- `activeConnectionCount++`; lokale `userId` und `deviceId` zunächst `null`.

- `redis.registerWs(ws)` mintet einen eindeutigen `wsId`. Der wird beim späteren `redis.registerClient` als Wert im `clients:{userId}`-Hash gespeichert, damit Cross-Node-Routing den Socket per `getLocalWs(wsId)` zurückfindet.
- `ws.on('message', async (raw, isBinary) => { ... })` - der Master-Handler:*Binär-Pfad (SRTP-Frames):**Text-Pfad:*
  - Nur für authentifizierte Verbindungen (`userId !== null`); Payload <= 8 KB.
  - `cid = wsActiveCall.get(ws)` - vorhanden nur nach `CALL_OFFER`/`CALL_ANSWER` (s. §4.11). Ohne Zuordnung wird der Frame verworfen.
  - `activeCalls.get(cid)` lokal, Fallback `redis.getActiveCall(cid)` (Cross-Node-Hydration).
  - **C3-Schutz:** `userId` muss `callerId` oder `calleeId` des Calls sein - sonst wird `wsActiveCall.delete(ws)` und der Frame ignoriert. Verhindert Frame-Injektion durch Dritte.
  - `peer = userId === callerId ? calleeId : callerId`, dann `await sendToUser(peer, raw)`. Bei Cross-Node-Peer wird der Buffer Base64-kodiert über Redis-Pub/Sub geschickt (s. §4.3).
  - `raw.length <= MAX_PAYLOAD (32 KB)` (sonst Close 1009 "Message too large").
  - `parseMessage(raw)` - gibt `null` bei kaputtem JSON: silent drop.
  - `msg.v !== PROTOCOL_VERSION` -> generischer `ERROR`.
  - Rate-Limit-Hooks am Anfang des Dispatch (`rateLimits` für authentifizierte User, `authRateLimits` für AUTH/REGISTER/INVITE_CREATE).
  - **Großer `switch(msg.type)`** mit ~30 Cases; jeder ruft einen spezialisierten `handle*`-Async-Handler auf.

3. `ws.on('close')` und `SESSION_END`:

- Lokales `clients`-Map-Cleanup; `await redis.unregisterClient(userId, deviceId); redis.unregisterWs(ws)`.
- `await isUserOnline(closingUserId)` (Redis-global): erst wenn der User auf **keiner** Node mehr verbunden ist, broadcast `STATUS_CHANGE`/`offline` und `userStatuses.delete`.
- `ipConnections` dekrementieren; `activeConnectionCount--`; `drainQüü()` zieht den nächsten Wartenden nach (s. §4.7).
- `pendingChallenges.delete(ws)` (falls Auth nie abgeschlossen wurde).

4. I2P-Listener (`REDE_I2P_PORT = 9378`):

- Separater `http.createServer()` + `new WebSocketServer({server: i2pServer})` an `127.0.0.1:9378`.
- Eingehende Verbindungen werden via `wss.emit('connection', ws, req)` an den Haupt-Handler durchgereicht - I2P-Clients gehen damit durch denselben Lifecycle wie Direct-WSS-Clients.
- Wird durch `REDE_NO_I2P=1` deaktiviert (z.B. auf Nodes, die nur über den Reverse-Proxy erreichbar sein sollen).

4.9 AUTH-FLOW UND MULTI-DEVICE-REGISTRIERUNG

REGISTER (neür User + erstes Gerät):

```
Client: { type: REGISTER, inviteCode, publicKey, signingKey,
          displayName, proof, deviceId? }
        proof = Ed25519.sign(utf8(displayName + publicKeyB64), signingSecretKey)

Server: 1. Formatvalidierung:
           - inviteCode = 32 Hex
           - displayName Regex (1-32, ohne Controls/'#')
           - publicKey, signingKey = 32-byte Base64
        2. proof verifizieren - bindet den Anzeigenamen an den Schlüsselbesitz
        3. invite.validateAndConsume(inviteCode) - atomar, single-use
        4. internalId = `${displayName}#${randomHex(2)}` (#xxxx-Suffix)
        5. store.addUser(internalId, displayName, publicKey, signingKey, deviceId)
        6. clients.set + await redis.registerClient(internalId, deviceId, ws)
        7. REGISTER_OK { userId, deviceId, serverSigningKey, deliveryToken }
```

AUTH (bestehender User):

```
Client: AUTH { userId, deviceId? }                  // Verbindung noch nicht auth'd

Server: handleAuthChallenge:
        - challenge = base64(crypto.randomBytes(32))
        - User/Gerät existiert NICHT?  -> trotzdem eine "fake"-Challenge zurück-
          geben (gleiche Form, gleicher RTT) und in pendingChallenges mit
          fake=trü merken - gegen User-Enumeration via Timing/Status.
        - AUTH_CHALLENGE { challenge }

Client: AUTH_RESPONSE {
          signature: Ed25519.sign(utf8("AUTH_CHALLENGE:" + challenge),
                                  signingSecretKey)
        }

Server: handleAuthResponse:
        - pendingChallenges.get(ws); 60s-Timeout, fake -> AUTH_FAIL (generisch)
        - signingKey = user.devices[deviceId].signingKey || user.signingKey
        - nacl.sign.detached.verify(utf8("AUTH_CHALLENGE:" + challenge),
                                    sig, signingKeyBytes)
        - clients.set + await redis.registerClient(userId, deviceId, ws)
        - userStatuses.set(...) + broadcastStatusChange('online')
        - sendAllStatuses(ws, userId)
        - AUTH_OK { userId, deviceId, serverSigningKey, prekeyCount, deliveryToken }
        - deliverPending(userId, ws, deviceId) - alle pendenden Nachrichten an dieses Gerät
```

Domain-Separation mit dem `"AUTH_CHALLENGE:"`-Präfix verhindert, dass eine vom Client signierte Challenge in einem anderen Protokollkontext (z.B. als Nachrichten-Header) wiederverwendet werden kann. Server und Client präfixieren beide den signierten String.

Multi-Device-Verknüpfung:

```
Bestehendes Gerät A  ->  Server  ->  Neüs Gerät B

DEVICE_LINK_CREATE
     ----------->
                       generiert one-time codeHash
                       (in device_links gespeichert, 5 min TTL)
                       <-- DEVICE_LINK_OK { code }
(User zeigt code dem neün Gerät)

                                          <-- DEVICE_LINK_USE {
                                                codeHash, publicKey, signingKey,
                                                deviceId,
                                                proof = Ed25519.sign(
                                                  utf8("DEVICE_LINK:"
                                                    + codeHash + ":"
                                                    + publicKey),
                                                  signingSecretKey)
                                              }
                       1. codeHash min 32 Hex (>=128 Bit Entropie) (M4-Audit)
                       2. store.useDeviceLink(codeHash)   // atomar, single-use
                       3. proof verifizieren
                       4. store.addDevice(targetUserId, newDeviceId,
                                         publicKey, signingKey)
                       5. clients.set + redis.registerClient
                       6. DEVICE_LINK_OK { userId, displayName, deviceId,
                                          serverSigningKey } ---->
DEVICE_ADDED {             sendToUser(targetUserId, DEVICE_ADDED, excludeDeviceId=B)
  deviceId, publicKey,
  signingKey }
     <-----------
```

Multi-Device-Fan-out: `redis.sendToUser(to, msg)` (s. §4.3) liefert an alle online verbundenen Geräte des Empfängers. Offline-Geräte (`!getOnlineDeviceIds(to).includes(devId)`) bekommen die Nachricht via `store.addPendingMessage(to, blob, ttl, MAX_PENDING_PER_USER=100, deviceId)` in eine **separate** Qüü pro Gerät (kein Cross-Device-Leakage).

Mit `toDeviceId` im Payload (gerätegezielter Versand, etwa für X3DH-First-Contact): `redis.sendToDevice(to, deviceId, msg)` matcht genau einen Eintrag im `clients:{to}`-Hash.

4.10 PRE-KEYS, OTP-RESERVIERUNG UND SEALED SENDER (SERVER-PFAD)

UPLOAD_PREKEYS:

```
Client: { type: UPLOAD_PREKEYS,
          signedPreKey, signedPreKeySig, oneTimePreKeys: [{id, pub}, ...],
          pqSignedPreKey?, pqSignedPreKeySig?, pqOneTimePreKeys? }
```

- `signedPreKeySig` = `Ed25519.sign(signedPreKey, identitySigningSecretKey)`. Server prüft die Signatur, bevor das Bundle field-level verschlüsselt in `pre_key_bundles` gespeichert wird (Audit-Fix K2).
- Optionaler PQXDH-Block: ML-KEM-768 Signed-Pre-Key + Signatur + One-Time-Bundles. Backward-kompatibel: läuft der Client klassisch, sind die `pq_*`-Felder einfach `NULL`. Schema-Migration `ALTER TABLE pre_key_bundles ADD COLUMN IF NOT EXISTS pq_*` läuft im pg-Store-Bootstrap automatisch (`pg-store.js:343`).

FETCH_PREKEY_BUNDLE + OTP-Reservierung:

```
Client A: FETCH_PREKEY_BUNDLE { for: targetId, deviceId? }
          |
Server:   1. prekeyFetchLimits (20/min pro Anfragender)
          2. otpConsumptionLimits (20/h pro Ziel-User - Anti-Enumeration)
          3. otpReservations[ rlHash(senderId + ':' + targetId + ':' + deviceId) ]:
             - Reservierung vorhanden?  -> dieselbe OTP zurückgeben (idempotent, 5 min)
             - sonst: nächste freie OTP für (targetId, deviceId) auswählen und mit
               expiresAt = now + 5 min reservieren
          4. PREKEY_BUNDLE { identityKey, signedPreKey, signedPreKeySig,
                             otpkId, otpkPub, pqSignedPreKey?, ... }
             - die OTP ist hier reserviert, aber NOCH NICHT KONSUMIERT
```

Die echte Konsumierung passiert **deferred**, beim ersten von A gesendeten `MESSAGE` mit `x3dh.usedOTPKId`:

```
handleMessage:
  if (x3dh && x3dh.usedOTPKId !== undefined && x3dh.usedOTPKId !== null) {
    await store.consumeOTP(to, toDeviceId || null, otpkId);   // atomar (SELECT FOR UPDATE + UPDATE)
    otpReservations.delete(rlHash(senderId + ':' + to + ':' + (toDeviceId || '')));
  }
  // Fallback für v2-Clients, die nur usedOTPKPub mitgeben:
  else if (x3dh.usedOTPKPub) {
    await store.consumeOTPByPub(to, toDeviceId || null, usedOTPKPub);
  }
```

Vorteile gegenüber Eager-Consume beim Fetch:

- Ein Angreifer, der nur Bundles abruft ohne Nachricht zu schicken, kann die OTP-Vorräte des Ziels **nicht** erschöpfen (zusammen mit dem Pro-Ziel-Stunden-limit aus Punkt 2).
- Doppelte Fetches desselben Senders innerhalb 5 min liefern dieselbe OTP -> keine OTP-Verschwendung durch Retries.
- Atomare DB-Operation verhindert die Race "zwei Clients fetchen, beide bekommen dieselbe OTP, beide konsumieren" (M3-Audit).

Sealed Sender - Server-Pfad:

```
Client: { type: SEALED_MESSAGE, to, sealedPayload, nonce, ttl,
          deliveryToken, toDeviceId? }
        sealedPayload = nacl.box(utf8("SEALED_SENDER_V1:" + JSON.stringify(inner)),
                                 nonce, recipientPubKey, ephemeralSecretKey)
        inner enthält den eigentlichen Absender + die verschlüsselte Nachricht

Server: 1. validateNonce + validateEncrypted + sealedRateLimits (30/min pro ipHash)
        2. verifyDeliveryToken(deliveryToken) - beweist: "irgendein authentifizierter
           User hat das gesendet" - ohne dass der Server die Identität kennt
        3. sealedPayload.ciphertext.length <= 32768 (s. §3.3)
        4. Routing wie normales MESSAGE: redis.sendToUser(to, ...) bzw.
           redis.sendToDevice(to, toDeviceId, ...), Offline-Qüü analog
        5. SEALED_MESSAGE_ACK - aber OHNE Absender-Korrelation
        6. Server-Logs enthalten NIE die senderId einer Sealed-Nachricht
```

Delivery-Token-Generierung und -Verifikation:

```
secret = _deliveryTokenSecret                  // 32 Byte; im Föderations-Modus aus Redis geteilt
ts     = floor(Date.now() / 1000)
token  = base64( UInt32BE(ts) || HMAC-SHA256(secret, UInt32BE(ts)) )    // 4 + 32 Byte

verify(tokenB64):
  buf = base64-decode(tokenB64)                // muss 36 Byte sein
  ts  = first 4 bytes (UInt32BE)
  hmac_received = next 32 bytes
  hmac_expected = HMAC-SHA256(secret, UInt32BE(ts))
  return timingSafeEqual(hmac_received, hmac_expected) && (now - ts < 86400)
```

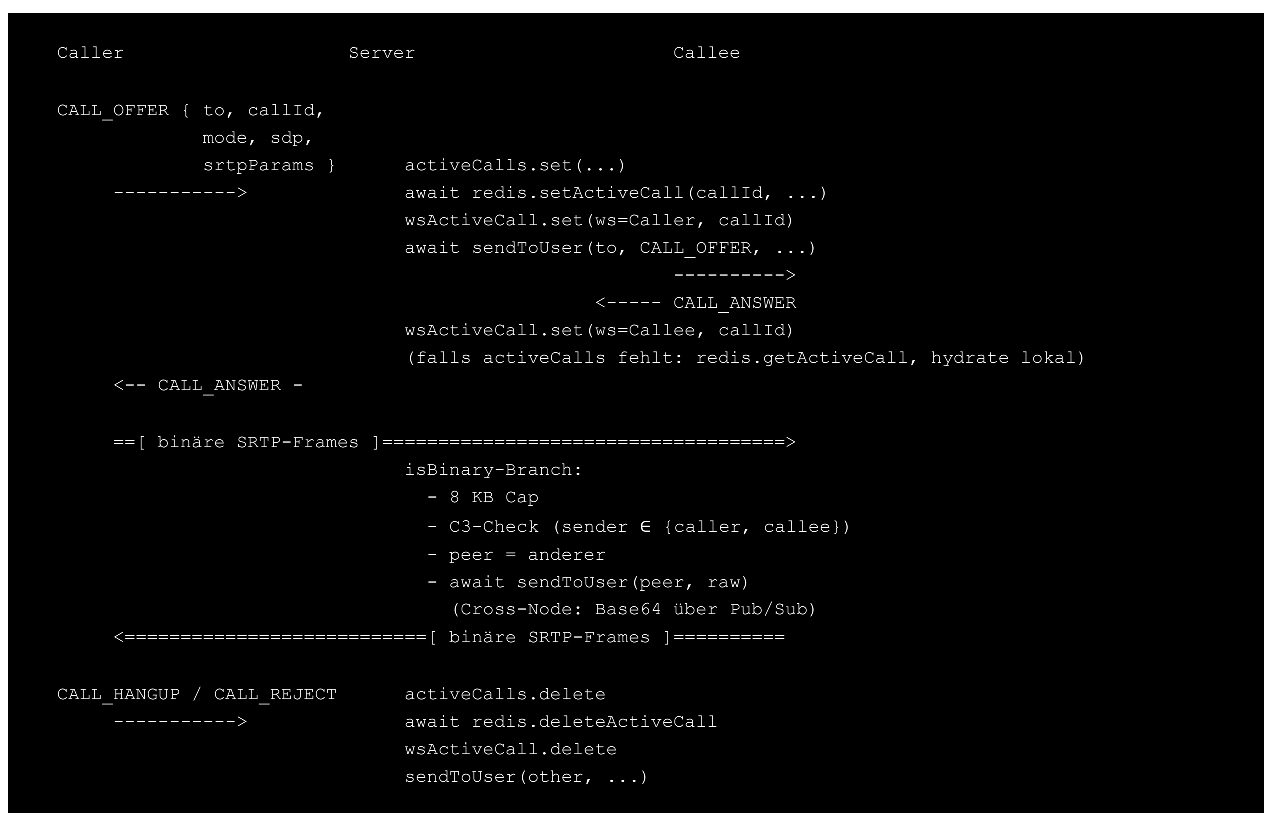
Tokens werden vom Server bei `AUTH_OK`/`REGISTER_OK` ausgegeben, sind 24 h gültig und fliessen NIE in den Sealed-Envelope (sonst lieferten sie wieder Absender-Korrelations-hinweise). In der Föderation teilen alle Nodes denselben `_deliveryTokenSecret` (Redis-Key `deliveryTokenSecret`, alle 5 min nachgezogen), damit ein auf Node A geprägter Token auch auf Node B validiert.

4.11 VOICE-CALL-SERVER (1:1 SRTP + GRUPPEN-LIVEKIT)

1:1-Anrufe - über WebSocket, **kein** SFU. Signaling als JSON, Audio als binäre SRTP-Frames im selben Socket.

State:

- `activeCalls: Map<callId, {callerId, calleeId, mode, startedAt}>` (lokal pro Node; im Föderationsmodus zusätzlich `activecall:{callId}` mit 4 h TTL in Redis, s. §4.3).
- `wsActiveCall: WeakMap<ws, callId>` (per-Verbindung-Zuordnung; lokal).
- **Caps:** max 500 aktive Calls global pro Node, max 2 Calls pro User. Rate-Limit: 20 Signaling-Messages/Min pro User (`callSignalingLimits`).



Whitelist im Relay-Payload (H2-Audit-Fix): nur `callId`, `to`, `from`, `fromDeviceId`, `mode`, `sdp`, `candidate`, `srtpParams`, `reason`, `muted`. Alle anderen Felder, die der Client schicken könnte, werden verworfen - der Server filtert.

CALL_BUSY, **CALL_RINGING**, **CALL_ICE**: reiner Relay ohne State-Änderung.

CALL_MUTE (in `handleSfuControl`): liest `activeCalls.get(callId)` mit Fallback `redis.getActiveCall(callId)`; relay des Mute-Status an die andere Partei.

Gruppen-Anrufe (`gcall_*`) - nutzen einen externen LiveKit-SFU + clientseitiges SFrame für E2EE. Server-Aufgaben:

- `gcallRoomName(scope) = HMAC-SHA256(LIVEKIT_API_SECRET, scope.kind + ':' + scope.id + (channelId ? ':' + channelId : '')).slice(0, 32)`. Deterministisch -> alle Teilnehmer eines Scopes landen im selben Raum. Per HMAC verschlüsselt -> der LiveKit-Operator kann Raum-IDs nicht auf Places/Channels zurückmappen.
- `GCALL_REQÜST_TOKEN`: `gcallTokenLimits` (10/min). Server signiert ein LiveKit-Access-Token (JWT, 6 h TTL) mit `room=gcallRoomName(scope)` und `identity = gcallPseudonym(userId, callId)` - Per-Call-rotierendes Pseudonym, LiveKit sieht keine echten User-IDs. Antwort: `GCALL_TOKEN { token, url }`.
- `GCALL_ANNOUNCE` / `GCALL_END`: relayed an alle Place-/Group-Mitglieder, damit sie erfahren, dass ein Call läuft/endet.
- `activeGroupCalls: Map<roomName, {scope, startedBy, startedAt, participants:Set}>` (lokal, per Node; Cap 200 Räume).
- **Audio-Inhalt:** Server sieht nichts. SFrame verschlüsselt jeden Audioframe mit einem Per-Call-Schlüssel, der über die normale E2EE-Group-Messaging-Schicht (Sender Keys) verteilt wird. LiveKit ist ein dummer Frame-Forwarder.

4.12 SERVER-SIGNATUREN, NONCE-REPLAY, STRUKTURVALIDIERUNG

`signedMessage(type, payload)` - jede vom Server gesendete Nachricht wird signiert:

```
function signedMessage(type, payload = {}) {
  const signingKey = store.getServerSigningKey();
  const msg = { ...payload, v: PROTOCOL_VERSION, type, ts: Date.now() };
  delete msg.serverSig;
  const body = JSON.stringify(msg);
  const sig  = nacl.sign.detached(naclUtil.decodeUTF8(body), signingKey.secret);
  msg.serverSig = naclUtil.encodeBase64(sig);
  return JSON.stringify(msg);
}
```

Reihenfolge wichtig: `payload` wird ZÜRST gespreadet, dann werden `v`/`type`/`ts` gesetzt - so kann ein böswilliger Client nicht via Payload-Feld `type` überschreiben. Ein etwaiges `serverSig` aus dem Payload wird vor dem Signieren explizit gelöscht.

Stolperfalle bei der Client-Verifikation: Der Empfänger darf das `serverSig`-Feld NICHT durch JSON-Reparsen entfernen - Map-Key-Reihenfolge und Zahlenformat in JS können sich zwischen `parse` und `stringify` ändern, dann scheitert die Signatur. Statt-dessen wird das Feld per Regex aus dem **Roh-String** geschnitten (Implementierung siehe Client-Doku, RedeConnection / network.js).

Nonce-Replay-Schutz:

```
validateNonce(b64)              - muss exakt 24 Byte decodieren
nonceKey = SHA256(senderId + ':' + nonceB64).hex
store.checkNonce(nonceKey)      - trü: neu, Eintrag mit seen_at = now() abgelegt
                                - false: Duplikat -> ERROR "Duplicate nonce (replay rejected)"
Cleanup-Job löscht Einträge > 24 h
```

Die Hash-Speicherung verhindert, dass die Datenbank eine `(senderId, nonce)`-Korrelations-tabelle im Klartext führt (Audit-Fix H2 / M1).

Wire-Size-Caps: siehe Tabelle in §3.3 - `MAX_HEADER_SIZE=1024`, `MAX_X3DH_SIZE=4096`, Sealed-Cap 32768. Änderungen hier müssen mit den Padding-Buckets im Client (16382 max) abgestimmt sein.

Generischer Client-ERROR-Banner (v2.19.10): Client-Seite fängt jedes `MSG.ERROR`, für das kein Service einen Handler registriert hat, und zeigt es als `[Server] <text>`-Toast. Server-Validierungsfehler sind seither sichtbar - wer hier neü Reject-Pfade hinzufügt, sollte aussagekräftige `error`-Texte vergeben.

Sender-Key-Signaturprüfung (`handleGroupMessage`, Sender-Key-Distribution): Der Server verifiziert `Ed25519.verify(utf8("SENDERKEY:" + groupId + ":" + chainKey + ":" + messageNumber), signature, senderSigningKey)`. Eine Sender-Key-Verteilung ohne gültige Signatur wird abgelehnt - verhindert, dass ein kompromittierter oder fehlerhafter Client Phantom-Sender-Keys einschleust (Audit-Fix M2/M3/H2).

Gruppen-Mitgliedschaftsprüfung: Server lehnt `GROUP_MESSAGE` ab, wenn `senderId` nicht in `group.members` ist - früher konnten leere Mitgliederlisten den Check überspringen (Audit-Fix H2).

4.13 LIFECYCLE, CLEANUPS, INVITES, STATUS

Startup-Reihenfolge (`main()` in `index.js`):

```
1. askPassphrase()                                    // stdin / REDE_SERVER_PASS
2. Passphrase-Stärke (>= 12 Zeichen, mind. 2 Zeichentypen)
3. await store.init(passphrase)                       // sqlite oder pg
                                                      // pg: advisory-lock-bootstrap, s. §4.3
4. Admin-CLI-Shortcuts (--gen-invite, --show-admin-key) - exit nach Druck
5. ADMIN_KEY = store.loadAdminKey()                   // gecached
6. redis.init({ workerId: REDE_WORKER_ID || REDE_NODE_ID })
7. await redis.connect()                              // No-op wenn REDE_REDIS_URL nicht gesetzt
8. redis.onCrossWorkerMessage(handler)                // Cross-Node-Inbound-Pub/Sub
9. _deliveryTokenSecret: get-or-set aus Redis (Föderation) bzw. crypto.randomBytes(32)
10. TLS oder Plain WS (certs/ prüfen) -> http(s).createServer
11. wss = new WebSocketServer({ server, path:'/rede', maxPayload: 32 KB })
12. wss.on('connection', ...) installieren - s. §4.8
13. server.listen(PORT, REDE_BIND || '127.0.0.1')
14. Optional: I2P-Listener (REDE_NO_I2P kontrolliert) auf 127.0.0.1:9378
15. Interaktive stdin-Konsole (invite/status/quit), falls TTY
16. Periodischer Cleanup-Loop alle 60 s starten
```

Periodische Aufräumarbeiten (60-s-Intervall):

Was	Bedingung
Alle ~20 Rate-Limit-Maps	`now - windowStart > 2 * window`
`activeCalls`	Calls älter als 4 h
`ipConnections`	Counter <= 0
`pendingChallenges`	Challenge älter als 60 s
`userStatuses`	User nicht (mehr) im lokalen `clients`
`otpReservations`	`now > expiresAt`
`activeGroupCalls`	Raum älter als 4 h ODER alle Phantom-Teilnehmer entfernt

Graceful Shutdown (`SIGINT`/`SIGTERM`):

```
cleanupOnExit():
  store.flushSync?.()         // sqlite: WAL-Checkpoint; pg: no-op
  store.zeroMasterKey?.()     // Master-Key mit Nullen überschreiben (best-effort)
  redis.shutdown()            // unsubscribe + DEL worker-alive:{id} + disconnect
process.exit(0)
```

`redis.shutdown` löscht den eigenen Heartbeat-Key sofort, damit Peers nicht erst 30 s warten müssen, bis die Stale-Cleanup-Logik anschlägt.

Invite-Codes - Lifecycle:

- `store.createInviteCode()` = `crypto.randomBytes(16).toString('hex')` (32 Hex-Zeichen). Server speichert nur `SHA256(code)` in `invite_codes` - der Klartext-Code geht nie auf Disk.
- `store.useInviteCode(code)` = `hashen; UPDATE invite_codes SET used=TRÜ WHERE code_hash=$1 AND used=FALSE`. Atomar, single-use, Race-safe.
- **Admin-CLI:** `node server/index.js --gen-invite` druckt einen Code und exit. Praktisch für Bootstrap-Einladungen ohne laufenden Server.
- **Netzwerk-Pfad** (`INVITE_CREATE`): Client liefert `adminKey`. Server vergleicht via `crypto.timingSafeEqual` nach SHA256-Hashing (gleichlange Buffer -> kein Length-Oracle). Rate-Limit 10/Stunde pro `adminKey`-Hash (`inviteRateLimits`). Bei Erfolg: `INVITE_CREATE_OK { code }` (server muss `await invite.generateInvite()` - siehe Audit/Fix-liste).
- **`invite-client.js`** ist ein kleiner Node-Client, der lokal den Admin-Key aus `data/.admin_key` liest und `INVITE_CREATE` über das WS-Protokoll schickt. Damit funktioniert das auch im pg-Föderations-Modus - die Codes landen am laufenden Server (also in der geteilten DB).

Geräte-Verknüpfungs-Codes (`device_links`-Tabelle):

- `store.createDeviceLink(userId, codeHash, expiresAt=5min)` - one-time, 5 min TTL.
- `store.useDeviceLink(codeHash)` - atomar, single-use, abgelaufene werden verworfen.
- `codeHash` muss mindestens 32 Hex-Zeichen (128 Bit) lang sein, sonst Reject (Audit-Fix M4).

Status-Broadcast - privacy-freundlich:

- `STATUS_UPDATE { status: 'online'|'away'|'dnd'|'invisible', customStatus? }`.
- `userStatuses.set(senderId, {status, customStatus})`.
- `broadcastStatusChange` verteilt `STATUS_CHANGE` an **alle online verbundenen User** (lokal). Der Server kennt keine Kontaktlisten - die Filterung passiert clientseitig.
- `invisible` wird über die Leitung als `offline` gesendet (interner Status bleibt korrekt; nach außen sieht der Empfänger nichts).
- Bei `AUTH_OK`: `sendAllStatuses(ws, excludeUserId)` schickt dem frisch angemeldeten User einen Snapshot aller derzeit sichtbar online-stehenden Users.

In der Föderation ist die Status-Sichtbarkeit per-Node best-effort: jeder Node sieht nur seine eigenen lokal verbundenen User. Cross-Node-Statusbroadcast ist ein dokumentierter Folgeschritt (s. §4.3 - "Was bleibt pro Node lokal").

5. DB-Schema (was der Server speichert)

Alle in der Tabelle aufgeführten Felder sind **field-level verschlüsselt** mit `nacl.secretbox(field, random_nonce, masterKey)` - der Master-Key kommt aus `scrypt(REDE_SERVER_PASS, master_salt, N=2^20)`. PostgreSQL/SQLite sieht nur Chiffretext.

Tabelle	**Spalten (Auszug)**	**Inhalt**
`users`	`internal_id` (PK), `display_name`, `public_key`, `signing_key`, `created_at`	Pro registriertem User; `display_name#xxxx`-Tag im PK
`devices`	`(user_id, device_id)` (PK), `public_key`, `signing_key`, `created_at`	Pro Gerät eines Users (X25519- + Ed25519-Pubkey)
`pre_key_bundles`	`bundle_key` (= `userId:deviceId`), `signed_pre_key`, `signed_pre_key_sig`, `one_time_pre_keys`, `pq_signed_pre_key`, `pq_signed_pre_key_sig`, `pq_one_time_pre_keys`, `updated_at`	Pro Gerät ein Bundle; PQ-Felder seit v2.19.0

Tabelle	Spalten (Auszug)	Inhalt
`pending_messages`	`id`, `recipient_key` (`userId:deviceId`), `blob`, `created_at`, `ttl`, `group_id?`, `place_id?`	Qüü für Offline-Empfänger; `blob` ist der vollständige Wire-Payload als JSON
`groups`	`group_id`, `name`, `created_at`	E2EE-Gruppen-Metadaten
`group_members`	`(group_id, user_id)`	Mitgliederliste
`places`	`place_id`, `creator_id`, `created_at`	Discord-ähnliche Server (Metadaten clientseitig verschlüsselt, hier nur Roster)
`place_members`	`(place_id, user_id)`	Mitgliederliste
`place_admins`	`(place_id, user_id)`	Admin-/Owner-Liste für Rollen-Enforcement
`place_channels`	`(place_id, channel_id)`	Kanal-IDs (Namen liegen E2EE im Client)
`invite_codes`	`code_hash`, `used`, `created_at`	nur SHA256 des Klartextcodes
`device_links`	`link_key`, `user_id`, `created_at`, `expires`	One-Time-Codes, 5 min TTL
`nonces`	`nonce_key`, `seen_at`	Replay-Schutz, 24-h-TTL, `nonce_key = SHA256(senderId:nonce)` (kein Klartext)
`blobs`	`blob_id`, `data`, `size`, `uploaded_at`, `user_hash`	Verschlüsselte Anhänge (max 10 MB, 30-Tage-TTL)
`_meta`	`key`, `valü`	`master_salt`, `passphrase_check`, `server_signing_key`, `admin_key` - gemeinsame Server-Identität für Föderation

Wichtig: Beim Editieren der Schemata muss `migrate-sqlite-to-pg.js` ggf. nachgezogen werden (z.B. der `pg-store.js`-Bootstrap fügt `pq_*`-Spalten via `ALTER TABLE ADD COLUMN IF NOT EXISTS` selbstständig hinzu, aber neü Tabellen erfordern explizite Migration).

6. Sicherheitsmodell (Server-Sicht)

Schicht	**Was der Server sieht**	**Was er nicht sieht**
Nachrichteninhalt	Chiffretext-Blob (`encrypted`)	Klartext (E2EE)
Absender (1:1)	Bei `MESSAGE`: `senderId` via Auth. Bei `SEALED_MESSAGE`: nur die nacl.box-Hülle	Bei Sealed: niemand (Sealed Send
Empfänger	`to` field	--
Nachrichtengröße	Frame-Länge	Padding macht alle Buckets gleich groß (256/1024/4096/16384)
Gruppen-Mitgliedschaft	Roster in `group_members`	Gruppen-Schlüssel, Inhalt
Place-Metadaten (Name, Channel-Namen, Topics, Emotes)	--	E2EE im Client mit MetadataKey
Audio-Inhalt	SRTP-Frames (verschlüsselt) / SFrame-Pakete	Audio-Klartext
IP-Adresse	bei Direct-WSS / I2P sieht nginx evtl. die IP, der WS-Server hängt die Hash-Form in `ipConnections`	Klartext-IP in `rateLimits`-Map-Keys (gehashed)
Sozialer Graph	nichts - Status-Broadcasts gehen an alle or Stehenden	Kontaktlisten der Clients

Validierung gegen kompromittierte Clients:

- `signedPreKeySig` muss verifizieren, bevor das Bundle gespeichert wird.
- `senderKeyHeader.signature` muss verifizieren, bevor `GROUP_MESSAGE` weitergeleitet wird.
- `proof` bei `REGISTER` / `DEVICE_LINK_USE` muss verifizieren, bevor irgendetwas persistiert wird.
- Strukturvalidierung (§3.3) läuft VOR jeder Krypto-Operation - gegen Heap-Erschöpfung und Format-Probing.

- Rate-Limits gegen Brute-Force und User-Enumeration (`targetAuthLimits`, OTP-Reservation, Sealed-Token-Format).

Server-Identität (Ed25519-Signing-Key):

- Wird einmal generiert (`_meta.server_signing_key` bzw. `data/.server_signing_key` im Pre-Federation-Modus) und sollte nie wechseln. Ein Wechsel sperrt alle Clients aus, deren TOFU-Pin den alten Public-Key hält - nur manüller Reset des Pins (`~/.rede/.cert_pin` / `serverSigningKey`-Profileintrag) hilft dann.
- Bei Föderation wird derselbe Schlüssel aus `_meta` geteilt -> ein Client kann transparent zwischen Nodes wechseln, ohne den Pin zu invalidieren.

7. Deployment

7.1 EINZELKNOTEN

systemd-Service (`server/rede-server.service`):

```
[Service]
ExecStart=/usr/bin/node /path/to/server/index.js
Restart=always
RestartSec=5s
StartLimitBurst=10
StartLimitIntervalSec=300
# Härtung
NoNewPrivileges=trü
ProtectSystem=strict
PrivateTmp=trü
ReadWritePaths=/home/amke/Rede/data /home/amke/Rede/server
User=amke
```

`User=` kontrolliert, ob `sudo systemctl restart rede-server` zum Neustart nötig ist (Antwort: ja, auch wenn der Service-User der eingeloggte User ist - systemctl selbst braucht root). Auf node-b läuft der Service als `maintainer`; dort genügt `kill -TERM $(pgrep -f "node server/index.js")` - systemd hängt mit `Restart=always` automatisch nach.

Umgebungsvariablen (`server/.env`):

```
REDE_SERVER_PASS=<server-passphrase>
REDE_DB_BACKEND=sqlite|pg
REDE_PG_URL=postgres://...
REDE_REDIS_URL=redis://...
REDE_LIVEKIT_URL=wss://...
REDE_LIVEKIT_API_KEY=...
REDE_LIVEKIT_API_SECRET=...
REDE_DEBUG_USER=<userId>            # optional, für dlog()-Tracing
```

Die Passphrase wird via `EnvironmentFile=-/home/amke/Rede/server/.env` an den Service durchgereicht; ohne sie prompts der Server interaktiv auf stdin, was unter systemd nicht funktioniert.

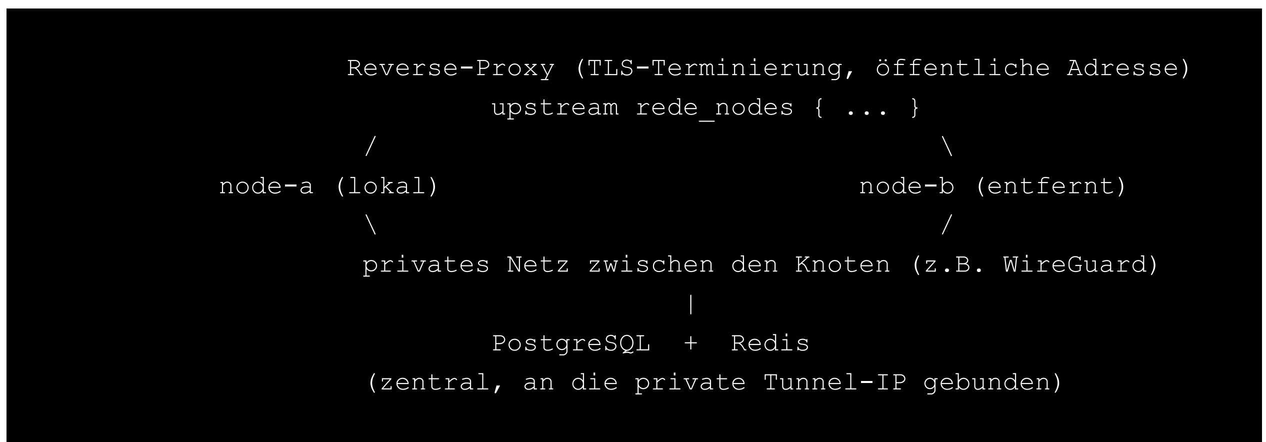
Direct WSS via nginx:

- nginx terminiert TLS auf `clip.jetzt`.
- Server läuft plain WS auf `127.0.0.1:9377`.
- WebSocket-Pfad: `/rede`.
- Optional: I2P-Endpoint auf `127.0.0.1:9378`, über den i2pd-Tunnel an die `.b32.i2p`-Adresse exponiert.

7.2 FÖDERIERTES DEPLOYMENT (MULTI-HOST-CLUSTER)

Für einen Mehr-Knoten-Cluster mit Lastverteilung und Failover gilt zusätzlich zum Einzelknoten-Deployment Folgendes. Architekturgrundlagen stehen in §4.3.

Topologie



Voraussetzungen

- **Privates Netz** zwischen allen Knoten (WireGuard, VPN o.ä.). PostgreSQL und Redis dürfen nicht öffentlich erreichbar sein - Bind an die private Tunnel-IP, nicht `0.0.0.0`. (Docker manipuliert iptables direkt und umgeht damit ufw; daher IP-gebundener Bind statt Firewall-Regel.)
- **PostgreSQL-Instanz**, an die private Tunnel-IP gebunden.
- **Redis-Instanz mit Passwort**, ebenfalls an die Tunnel-IP gebunden.
- **Reverse-Proxy** am öffentlichen Endpunkt mit Round-Robin-`upstream`-Block auf die internen `:9377`-Ports der Knoten.

Erstinstallation (Cluster aus einer bestehenden Single-Node ausweiten)

1. PostgreSQL + Redis starten (z.B. via `infra/docker-compose.yml`, Ports an die Tunnel-IP gebunden).

2. Bestehende SQLite-DB nach PostgreSQL migrieren: `node server/migrate-sqlite-to-pg.js`. Das Skript ist read-only auf SQLite und transaktional auf PostgreSQL.

3. `server/.env` auf dieser Node setzen:REDE_DB_BACKEND=pg
REDE_PG_URL=postgresql://rede:<PW>@<tunnel-ip>:5432/rede
REDE_REDIS_URL=redis://:<PW>@<tunnel-ip>:6379
REDE_NODE_ID=node-a

4. **`data/` nicht löschen** - `.master_salt`, `.server_signing_key`, `.admin_key` werden beim ersten pg-Start automatisch in `_meta` importiert. So bleibt die Server-Identität (Ed25519-Signing-Key, der TOFU-Pin der Clients) erhalten.

5. systemd-Restart; das Log meldet `Federation enabled - worker node-a`.

Weitere Knoten beitreten lassen

1. Repository klonen (`git clone <deploy-repo>`), `./install.sh` oder `npm install`.

2. `server/.env` mit **identischer** Passphrase und identischen PG-/Redis-URLs wie der Hauptknoten:REDE_SERVER_PASS=<gleiche Passphrase wie auf node-a>
REDE_DB_BACKEND=pg
REDE_PG_URL=postgresql://rede:<PW>@<tunnel-ip>:5432/rede
REDE_REDIS_URL=redis://:<PW>@<tunnel-ip>:6379
REDE_NODE_ID=node-b
REDE_BIND=<eigene-tunnel-ip>
REDE_TRUSTED_PROXIES=<reverse-proxy-ip>
REDE_NO_I2P=1

3. `data/` leer lassen - Signing-Key/Salt/Admin-Key werden aus `_meta` gezogen.

4. Starten und im Log auf `Federation enabled - worker node-b` prüfen.

5. Im Reverse-Proxy einen Eintrag für den neün Knoten zum `upstream`-Block hinzufügen.

nginx als Lastverteiler

```
upstream rede_nodes {
    server 127.0.0.1:9377 max_fails=3 fail_timeout=10s;     # node-a (lokal)
    server 10.99.0.2:9377 max_fails=3 fail_timeout=10s;     # node-b (über Tunnel)
}

location /rede {
    proxy_pass http://rede_nodes;
    proxy_http_version 1.1;
    proxy_set_header Upgrade $http_upgrade;
    proxy_set_header Connection "upgrade";
    proxy_set_header Host $host;
    proxy_read_timeout 86400;
    proxy_send_timeout 86400;
}
```

`max_fails`/`fail_timeout` liefern passive Health-Checks: fällt eine Node aus, routet nginx neü Verbindungen nach drei Fehlversuchen ausschliesslich zur anderen - ohne weitere Konfigurations-änderung.

HAProxy als zusätzliche Schicht ist nicht erforderlich; wenn vorhanden, läuft er hinter nginx (plain auf einem internen Port wie `127.0.0.1:8080`), nicht als zweiter öffentlicher TLS-Eingang - sonst müssten Clients umkonfiguriert und Zertifikate neu gepinnt werden.

Failover-Verhalten

- **Crash einer Node:** Der Heartbeat-Key `worker-alive:{id}` läuft nach 30 s ab; lebende Nodes räumen die zugehörigen `clients:{uid}`-Einträge automatisch über `getClientDevices`/`HDEL` auf (Self-Healing).
- **Reverse-Proxy** markiert die Node bei wiederholten Verbindungsfehlern als down und vermeidet sie während `fail_timeout`.
- **Clients** mit Verbindung zur ausgefallenen Node reconnecten dank exponentiellem Backoff im Client und landen über den Reverse-Proxy auf einer gesunden Node - mit derselben Server-Identität (TOFU intakt).

Sicherheitsmodell unter Föderation

- **Server-Signaturschlüssel** wird einmal generiert und via `_meta` geteilt; alle Knoten signieren mit demselben Ed25519-Schlüssel. Der TOFU-Pin der Clients bleibt gültig.
- **Nachrichten** reisen zwischen Knoten als bereits signierte und Ende-zu-Ende-verschlüsselte Payloads durch Redis Pub/Sub - Redis sieht keine Klartextinhalte.
- **DB-Felder** bleiben field-level verschlüsselt (Master-Key aus Passphrase + Salt); PostgreSQL sieht ebenfalls keinen Klartext.
- **Sealed-Sender-Tokens** werden über `deliveryTokenSecret` in Redis geteilt, damit ein auf Node A geprägter Token auch von Node B akzeptiert wird.
- `REDE_TRUSTED_PROXIES` darf nur die IPs der vertraünswürdigen Reverse-Proxies enthalten - diese Verbindungen werden vom Per-IP-Cap ausgenommen.

Bekannte Einschränkungen

- **Per-Node-Rate-Limits** (s. §4.3): können sich mit N Knoten faktisch ver-N-fachen. Akzeptabel; Migration auf `redis.checkRateLimit` ist ein Folgeschritt.
- **Per-Node-Statusbroadcast:** Presence wird nur an lokal verbundene Clients gesendet.
- **I2P-Endpoint:** lebt nur auf einem Knoten (eine I2P-Adresse = ein Tunnel-endpunkt). Cross-Node-Zustellung von und zu I2P-Clients funktioniert dennoch durch die Föderation.
- **Daten-Schicht-SPOF:** PostgreSQL und Redis laufen typischerweise auf einer Maschine. Echte HA auf Datenebene (Postgres-Replikation, Redis Sentinel) ist ein Folgeschritt jenseits dieses Layouts.

Vollständiger Rollout

Eine real durchgeführte Federation-Inbetriebnahme inklusive Vorfällen und Reparaturen ist separat dokumentiert in [`FEDERATION-DEPLOYMENT.md`].

7.3 PATCH-WORKFLOW AUF ZWEI NODES

Kanonischer Code liegt auf node-a unter `/home/amke/Rede/server/`. Bei einer Server-Änderung:

1. **Edit** `/home/amke/Rede/server/index.js` (oder die andere betroffene Datei) auf node-a.
2. **Mirror ins Deploy-Repo:** `cp` ins lokale `rede-server-repo/`, `git commit`/`git push` zu `caaatto/rede-server` (privat).
3. **Mirror nach node-b:** `scp /home/amke/Rede/server/index.js maintainer@10.99.0.2:/home/maintainer/rede-server/server/index.js` (node-b's HTTPS-Remote auf `caaatto/rede-server` kann ohne Credentials nicht ziehen - direkter scp ist der zuverlässige Weg).

4. **Restart node-a:** `sudo systemctl restart rede-server` (Service läuft als User `amke`, systemctl-Aufruf braucht sudo).

5. **Restart node-b:** `ssh maintainer@10.99.0.2 'kill -TERM $(pgrep -f "node server/index.js")'` - Service läuft dort als `maintainer`, systemd hat `Restart=always`, der Prozess kommt mit neüm Code zurück (sudo-frei).

6. **Verifikation:** beide Prozess-PIDs neu, `worker-alive:node-a` + `worker-alive:node-b` in Redis vorhanden, `PUBSUB NUMSUB worker:node-a worker:node-b` jeweils `1`.

Live-Diagnose ohne Code-Änderung: `/tmp/rede-pubsub-tap.js` (siehe §4.3 Bekannte Edge-Cases) - subscribt `worker:*` und loggt jede Cross-Node-Message mit `action`, `wsId`, abgeleitetem `innerType` und Byte-Länge. Read-only, hat keinen Einfluss aufs Routing.

8. Abhängigkeiten

SERVER (NODE.JS)

Paket	Version	Verwendung
ws	8.19.0	WebSocket-Server
tweetnacl	1.0.3	NaCl-Kryptografie
tweetnacl-util	0.15.1	Base64/UTF8-Utilities
better-sqlite3	^12.8.0	SQLite-Datenbank
pg	^8.20.0	PostgreSQL-Client
ioredis	^5.10.1	Redis-Client
mediasoup	^3.19.19	SFU für Sprachanrufe (Legacy-Pfad)
socks-proxy-agent	8.0.5	I2P/Tor-Proxy (im Server selten genutzt, hauptsächlich Client)

INFRASTRUKTUR (FÖDERATION)

Komponente	Version	Verwendung
PostgreSQL	16	gemeinsamer Datenspeicher
Redis	7	Pub/Sub-Routing + Shared State
nginx	1.20+	TLS-Terminierung + Round-Robin-LB
WireGuard	--	privates Netz zwischen Nodes
LiveKit	--	SFU für Gruppenanrufe (externer Dienst, JWT-authentifizie

Stand: 2026-05-26.