Rede Client Technische Dokumentation

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

GEARFISH TEAM: KEANU AMANN | DAVID KLEINFERCHER

Dieses Dokument verschriftlicht ausschließlich die **Client-Seite** - den Avalonia-v2-Desktop-Client und den Legacy-Terminal-v1-Client. Die Server-Seite wird im Rede-Server-Teil behandelt. Das Wire-Protokoll wird in beiden Dokumenten beschrieben - hier aus der Sicht dessen, was der Client implementiert, sendet und auf eingehende Nachrichten handelt.

Inhalt

Rede Client Technische Dokumentation 1
Version: v2.19.11-beta | Protokollversion: 3 | Lizenz: AGPL-3.0 1
Inhalt 2
1. Projektstruktur 4
2. Architekturüberblick (Client-Sicht) 5
3. Kryptografie 6
3.1 Schlüsselgenerierung 6
3.2 X3DH / PQXDH 7
3.3 Double Ratchet 10
3.4 Sender Keys (Gruppen) 12
3.5 Sealed Sender 13
3.6 Nachrichtenpadding 14
3.7 HKDF-SHA256 14
3.8 Profilverschlüsselung (At-Rest) 15
3.9 Schlüsselzeroisierung 16
3.10 SRTP (Sprachanrufe) 17
3.11 SFrame (Gruppenanrufe) 18
4. Protokoll (Client-Sicht) 19
5. Client v2 - Desktop (Avalonia) 21
5.1 Projektstruktur 21
5.2 Rede.Core - Geschäftslogik 22
5.3 Rede.Desktop - UI 23
5.4 Dienste (Services) 25
5.5 Audio-Pipeline 26
5.6 Themen und Styling 27
6. Client v1 - Terminal (Node.js) 28
7. Datenmodell (im Client) 30
7.1 Profil 30
7.2 Kontakt 31
7.3 Gruppe 31
7.4 Place 32

7.5 Chatnachricht 33
8. Places - Discord-ähnliche Server 34
8.1 Kanalstruktur 34
8.2 Rollensystem 35
8.3 Berechtigungen 35
8.4 Metadaten-Verschlüsselung 36
9. Sprachanrufe 37
9.1 1:1 Anrufe (SRTP) 37
9.2 Gruppenanrufe (LiveKit + SFrame) 38
10. Transport und Netzwerk 40
10.1 Transportmodi 40
10.2 TOFU-Zertifikatspinning 41
10.3 Server-Signaturverifikation 42
10.4 Automatische Wiederverbindung 43
11. Sicherheitsmodell (Client) 44
11.1 Threat Model 44
11.2 Sichtbarkeit für den Server (Wiederholung aus Client-Perspektive) 45
11.3 Angewandte Sicherheitshärtungen 45
12. Build, Release und Auto-Update 47
12.1 Build-Prozess 47
12.2 Release-Prozess 47
12.3 Auto-Update 49
13. Abhängigkeiten 51
Client v2 (C# / .NET 8) 51
Native Libraries 51
Client v1 (Node.js) 52

1. Projektstruktur

Der Client lebt unter `rede-client/` und ist ein **öffentliches** Git-Repository: `git@github.com:caaatto/rede.git`, Branch `main`. Die Releases liegen unter https://github.com/caaatto/rede/releases.

```
rede-client/
|-- client/                       # v1 Terminal-Client (4784 Zeilen JS)
|   |-- index.js                  # TUI-Einstieg (1336 Zeilen)
|   |-- cli.js                    # CLI-Modus (1268 Zeilen)
|   |-- crypto.js                 # Kryptografie (1017 Zeilen)
|   |-- store.js                  # Profilspeicher (376 Zeilen)
|   |-- network.js                # WebSocket-Client (319 Zeilen)
|   |-- ui.js                     # Terminal-UI (265 Zeilen)
|   +-- boot.js                   # Startup (203 Zeilen)
|-- shared/
|   +-- protocol.js               # Protokolldefinitionen (Clientkopie,
                                    identisch zu /home/amke/Rede/shared/protocol.js)
|-- src/                          # v2 Desktop-Client (~18.600 Zeilen C#)
|   |-- Rede.Core/                # Geschäftslogik-Bibliothek
|   +-- Rede.Desktop/             # Avalonia-UI-Projekt
|-- scripts/
|   |-- build-release.sh / .ps1       # Release-Build
|   |-- sign-release.sh               # Ed25519-Signatur für Releases
|   |-- install.sh / .ps1             # End-User-Installer
|   |-- install-rnnoise.sh / .ps1     # RNNoise-Native-Lib-Installer
|   +-- build-rnnoise.sh              # RNNoise aus Qüllen kompilieren
|-- packaging/                    # MSIX-, Linux-.desktop-Dateien
+-- Rede.sln                      # .NET Solution
```

Codeumfang:

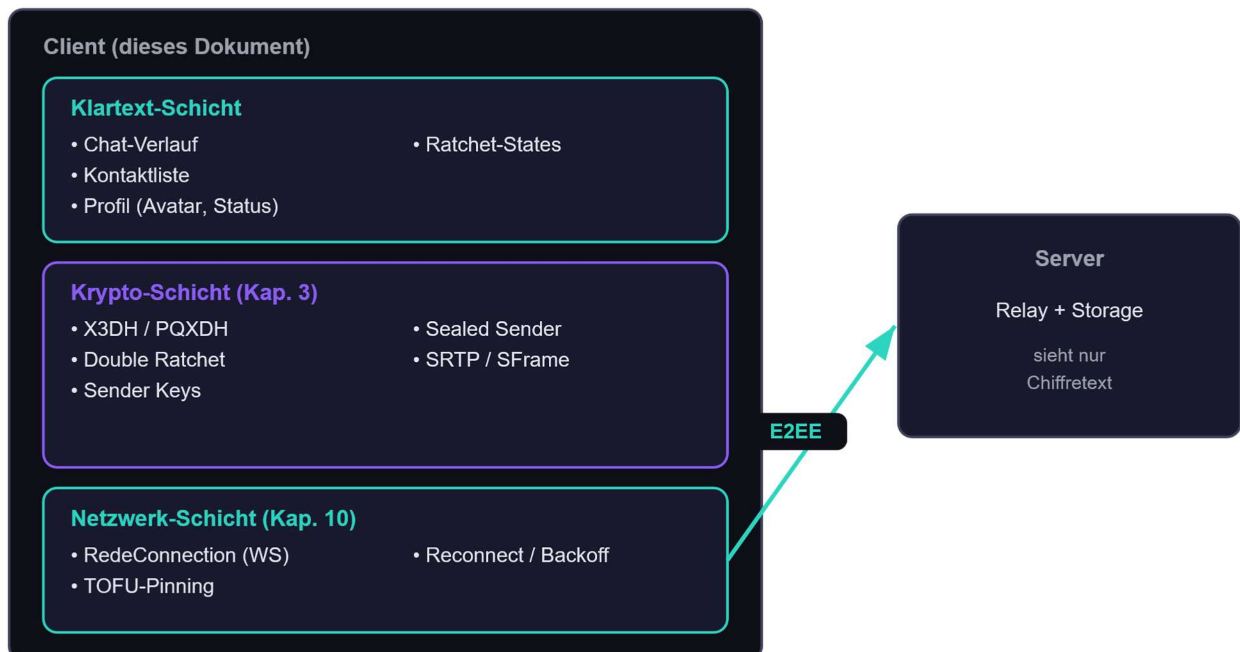
Komponente	Sprache	Zeilen
Client v1 (TUI+CLI)	JavaScript	~4.800
Client v2 (Rede.Core)	C#	~10.400
Client v2 (Rede.Desktop)	C#/AXAML	~8.200
Protokoll (shared)	JavaScript	147

Wichtig: `client/index.js` und `client/cli.js` (v1) sind parallele Implementierungen. Jeder Sicherheitsfix muss in BEIDE Dateien eingepflegt werden.

2. Architekturüberblick (Client-Sicht)

Der Client hält **alle Klartextdaten** lokal: Nachrichten, Profile, Kontaktlisten, Ratchet-States, Audio-Klartext beim Anruf. Der Server bekommt nur Chiffretext zu sehen. Vor jeder Wire-Operation läuft der Client durch die Krypto-Schicht (Kap. 3) und packt Nachrichten in das Protokollformat (Kap. 4).

Architekturüberblick — Client / Server



Vertraünsanker:

- **Ed25519-Server-Signing-Key** (TOFU-gepinnt pro Profil) - der entscheidende Identitätsbeweis für den Server.
- **TLS-Cert-Fingerprint** - ebenfalls TOFU-gepinnt, aber lockerere Policy (silent Re-Pin bei CA-validierten Rotations, s. §10.2).
- **Pre-Key-Bundles** der Kontakte mit Ed25519-signierten SPK - vor X3DH verifiziert.

Forward Secrecy + Post-Compromise Security durch Double Ratchet pro 1:1-Konversation und Sender-Key-Rotation in Gruppen. Vergangene und zukünftige Nachrichten bleiben sicher, selbst wenn der aktülle Schlüssel kompromittiert wird.

3. Kryptografie

3.1 SCHLÜSSELGENERIERUNG

Alle Schlüssel werden über libsodium (Sodium.Core NuGet) generiert:

Schlüsseltyp	Algorithmus	Grösse	Verwendung
Identität (Encryption)	X25519	32 Bytes	DH-Austausch, nacl.box
Identität (Signing)	Ed25519	64 Bytes (secret)	Signaturen, Fingerabdrücke
Signed Pre-Key	X25519	32 Bytes	X3DH, mit Ed25519 signiert
One-Time Pre-Key	X25519	32 Bytes	X3DH, einmalig
PQ Signed Pre-Key	ML-KEM-768	Pub 1184 B, Priv 2400 B	PQXDH, mit Ed25519 signiert
PQ One-Time Pre-Key	ML-KEM-768	dito	PQXDH, einmalig
Symmetrischer Schlüssel	Random	32 Bytes	Gruppen, Place-Metadaten

```
Datei: src/Rede.Core/Crypto/CryptoService.cs

GenerateKeyPair()          -> X25519 Keypair (PublicKeyBox.GenerateKeyPair)
GenerateSigningKeyPair()   -> Ed25519 Keypair (PublicKeyAuth.GenerateKeyPair)
GenerateSymmetricKey()     -> 32 Byte Zufall (SodiumCore.GetRandomBytes)
```

Schlüsselformat: In-Memory werden alle Schlüssel als `byte[]` gehalten, um sie nach Gebrauch nullen zu können. Auf dem Draht und in der Profildatei werden sie als Base64-Strings transportiert. Der `Base64BytesJsonConverter` sorgt für automatische Konvertierung bei Serialisierung/Deserialisierung. **Schlüssel-Vergleiche** verwenden konseqünt `CryptographicOperations.FixedTimeEquals` - nie `==`.

3.2 X3DH / PQXDH

```
Dateien: src/Rede.Core/Crypto/X3dh.cs (~270 Zeilen)
         src/Rede.Core/Crypto/PQKem.cs (ML-KEM-768 Wrapper, BouncyCastle.Cryptography 2.6.2)
```

X3DH etabliert einen initialen gemeinsamen Schlüssel zwischen zwei Parteien, die nie gleichzeitig online sein müssen. Das Protokoll basiert auf dem Signal-Entwurf.

Seit v2.19.0-beta: Hybrider Handshake nach Signal PQXDH-Spec (Mai 2024). Zusätzlich zu den klassischen X25519-Schlüsseln nutzen beide Parteien ML-KEM-768 (FIPS 203, ehemals Kyber768) Schlüssel. Der gemeinsame Schlüssel wird aus DH-Outputs **und** einem KEM-Shared-Secret abgeleitet - ein Angreifer muss **beide** Primitive brechen. Schützt gegen "Harvest now, decrypt later"-Angriffe mit zukünftigen Quantencomputern. Klassisches X25519 wird beibehalten falls der Peer keine PQ-Schlüssel hat (logged Warnung).

PQXDH HKDF-Input (vs. klassisch):

```
klassisch: HKDF(DH1 || DH2 || DH3 [|| DH4],          salt, "RedeX3DH")
PQXDH:     HKDF(DH1 || DH2 || DH3 [|| DH4] || SS_PQ, salt, "RedePQXDH")
```

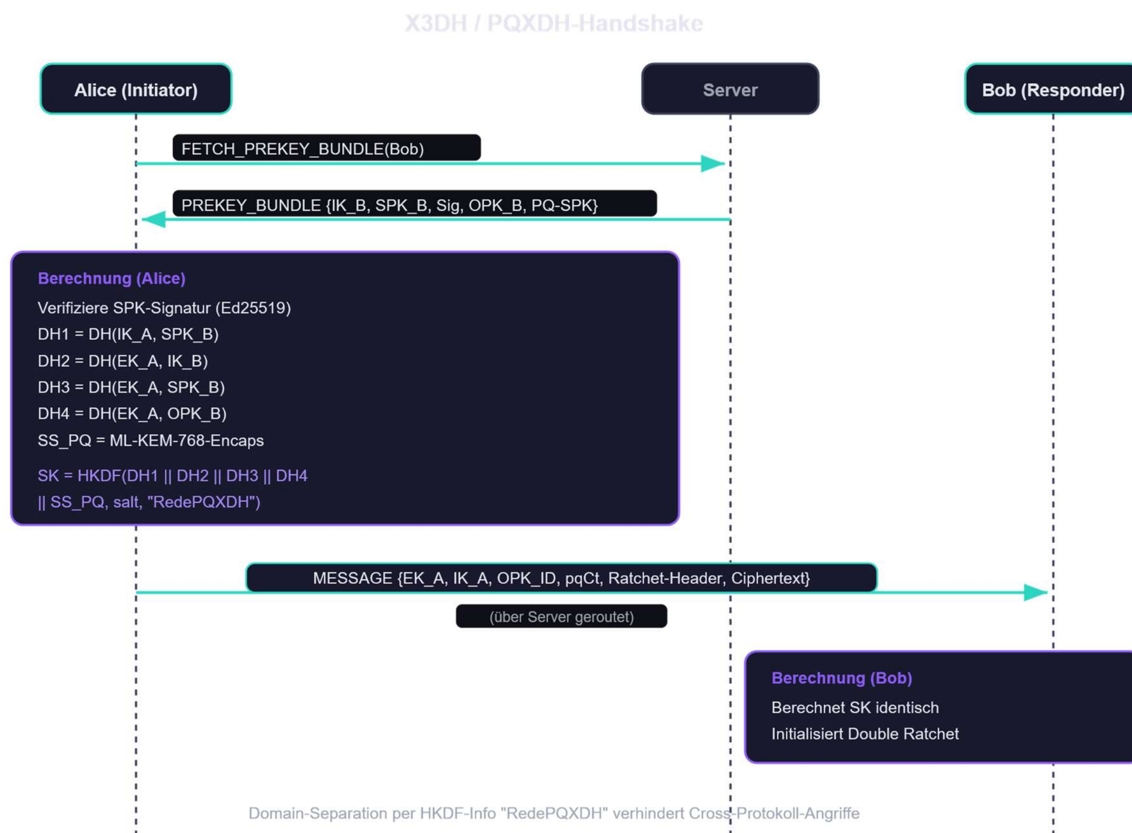
`SS_PQ` ist 32 Bytes aus ML-KEM-Encapsulation gegen Bobs PQ-OPK (bevorzugt) oder PQ-SPK (Fallback). Domain-Separation über HKDF-Info verhindert Cross-Protokoll-Angriffe.

ML-KEM-768 Schlüsselgrössen: Pub 1184 B, Priv 2400 B, Ciphertext 1088 B, Shared Secret 32 B. Pre-Key-Bundle wächst dadurch um ~1.5 KB, jede Erstnachricht um ~1 KB. Sender Keys (Gruppen) erben PQ-Schutz automatisch über den 1:1-Bootstrap-Channel.

Rollen:

- **Initiator (Alice)**: Will die erste Nachricht senden
- **Responder (Bob)**: Hat Pre-Key-Bündel beim Server hinterlegt

Ablauf:



HKDF-Salt für X3DH:

Der Salt wird aus beiden Identitätsschlüsseln abgeleitet, sortiert für Determinismus:

```
salt = SHA256("RedeX3DHSalt" || sorted(IK_A, IK_B))
```

Validierungen:

- SPK-Signatur wird mit Ed25519 verifiziert
- Alle DH-Schlüssel werden auf Low-Order-Punkte geprüft (7 bekannte Curve25519-Punkte)
- DH-Ausgabe wird auf All-Zeros geprüft
- Alle ephemeren Schlüssel werden in `try-finally` genullt

Wire-Grösse des x3dh-Blocks: Klassisches X3DH bringt nur `identityKey`, `ephemeralKey` und `usedOTPKPub` (je 32 B, base64 ~44 chars) - ~250 chars JSON insgesamt. PQXDH fügt `pqCt` (ML-KEM-768-Ciphertext, 1088 B roh, ~1452 base64) sowie optional `usedPQOTPKPub` hinzu, was den Block auf ~1700 chars JSON treibt. Das wird vom Server gegen `MAX_X3DH_SIZE = 4096` gehalten (siehe Server-Doku §3.3). Beim Erweitern des x3dh-Blocks (z.B. neüre PQ-Algorithmen) immer Headroom gegen diesen Cap behalten.

OTP-Verbrauch: Nach erfolgreichem Decrypt einer X3DH-Initial-Message löscht der Empfänger den verwendeten OTPK aus seinem Profil. Vorher nicht - bei einem fehlgeschlagenen Decrypt (z.B. crossed-resync) bleibt der OTPK verwendbar für den nächsten Versuch (v2.19.7-Fix).

Crossed-X3DH-Konvergenz: Initiieren beide Seiten gleichzeitig (z.B. nach gegenseitigem `/resync`), erzeugen beide einen eigenen Initiator-State und überschreiben sich gegenseitig. Ohne Eingriff fällt jede Folge-Nachricht auf eine andere Chain als der Peer erwartet - silent fail. Lösung in `ChatService.cs`: deterministischer Tiebreaker per `string.CompareOrdinal(self.UserId, from)`. Die "niedrigere" UserID behält ihren Initiator-State, die "höhere" adoptiert den Responder-State der anderen Seite (v2.19.8-Fix).

Archivierte SPKs: Beim Schlüsselrollover wird der bisherige `SignedPreKey` in `PreviousSignedPreKeys` (max 5 Einträge) verschoben. So lassen sich X3DH-Initials weiterhin entschlüsseln, die der Peer mit dem **vorigen** SPK gebaut hat. Gleiche Lifecycle für PQ-SPKs (v2.19.9-Fix).

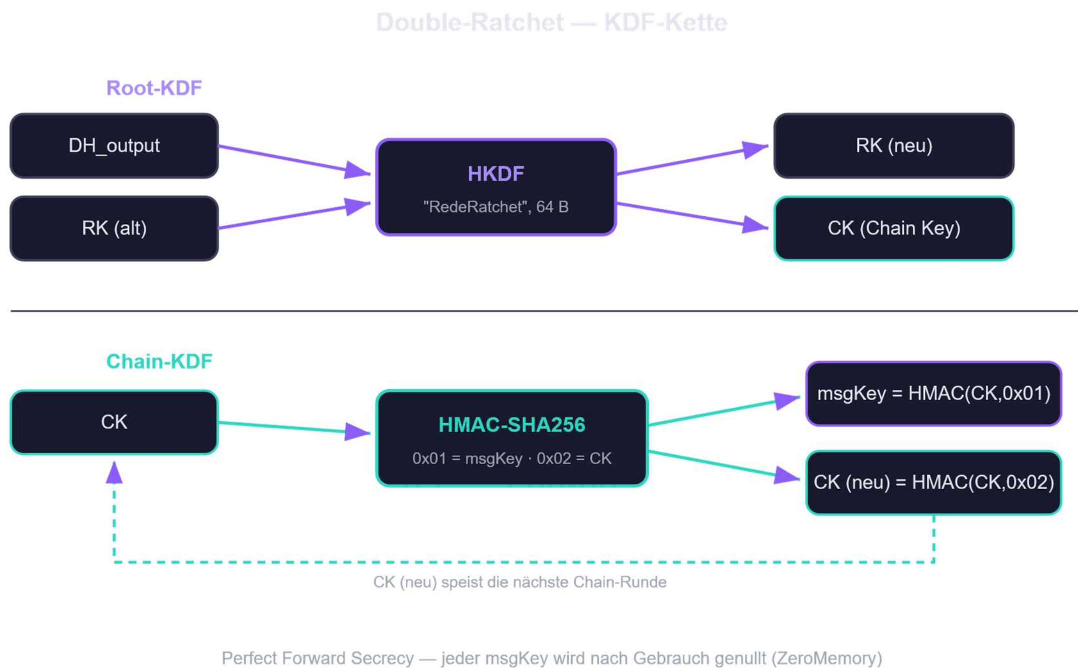
3.3 DOUBLE RATCHET

```
Datei: src/Rede.Core/Crypto/DoubleRatchet.cs (431 Zeilen)
```

Der Double Ratchet bietet Perfect Forward Secrecy pro Nachricht für 1:1-Gespräche. Jede Nachricht wird mit einem einzigartigen Schlüssel verschlüsselt.

Ratchet-Zustand (RatchetState):

Feld	Typ	Beschreibung
`DHs`	KeyPairBytes	Eigenes DH-Schlüsselpaar (rotierend)
`DHr`	byte[]	Empfangener DH-Public-Key
`RK`	byte[]	Root Key (32 Byte)
`CKs`	byte[]	Sending Chain Key
`CKr`	byte[]	Receiving Chain Key
`Ns`	int	Nachrichtenzähler (senden)
`Nr`	int	Nachrichtenzähler (empfangen)
`PN`	int	Vorheriger Sendezähler
`MKSKIPPED`	Dict	Übersprungene Nachrichtenschlüssel



Verschlüsselung:

```
1. Nachricht padden (MessagePadding)
2. MsgKey ableiten via KdfCK
3. Verschlüsseln mit XSalsa20-Poly1305 (nacl.secretbox)
4. Header: {dh: Base64(DHs.pub), pn: PN, n: Ns}
```

Sicherheitsgrenzen:

- `MaxSkip = 256`: Maximal 256 übersprungene Nachrichten
- `MaxMkSkipped = 1000`: Maximal 1000 gespeicherte Nachrichtenschlüssel
- `MaxMessageNumber = 1.000.000.000`: Zähler-Overflow-Schutz
- Ratchet-State-Rollback bei fehlgeschlagener Entschlüsselung (DeepClone + Restore)

Ratchet-State-Schlüssel im Profil: `Profile.RatchetStates[contactId:deviceId]`. Pro Empfänger-Device ein eigener State. **Legacy-Fallback:** Wenn `contact.Devices` leer ist (Alt-Kontakte, USER_LOOKUP_OK ohne `devices`-Feld), wird der State unter `Profile.RatchetStates[contactId]` (kein `:deviceId`) gespeichert. Sowohl `ChatService.SendMessageAsync` als auch `CallService.InitiateCall` (seit v2.19.11) fallen darauf zurück, wenn keine per-device States existieren.

3.4 SENDER KEYS (GRUPPEN)

```
Datei: src/Rede.Core/Crypto/SenderKeys.cs (155 Zeilen)
```

Sender Keys bieten eine effiziente symmetrische Verschlüsselung für Gruppen. Jeder Teilnehmer hat einen eigenen Sender Key, den er an alle anderen verteilt.

Zustand (SenderKeyState):

- `ChainKey`: 32 Byte symmetrischer Schlüssel
- `MessageNumber`: Zähler

Verschlüsselung:

```
1. MsgKey ableiten via HMAC-SHA256 (gleiche KdfCK wie Double Ratchet)
2. Nachricht padden
3. Verschlüsseln mit XSalsa20-Poly1305
4. Signatur: Ed25519(ciphertext || uint32(msgNumber) || UTF8(contextId))
```

Die `contextId` bindet die Signatur an eine bestimmte Gruppe oder einen Place-Kanal und verhindert gruppenübergreifende Replay-Angriffe.

Limits:

- `MaxSkip = 1000`
- `MaxMessageNumber = 10.000` (danach ist ein Rekey erforderlich)
- Signatur immer mit kontextgebundenen Daten verifiziert

Verteilung neuer Sender Keys: Beim Beitreten einer Gruppe verteilt jedes Mitglied seinen Sender Key über die bestehende 1:1-Double-Ratchet-Verbindung zu jedem anderen Mitglied. Der Server prüft die Ed25519-Signatur über `SENDERKEY:{groupId}:{chainKey}:{messageNumber}`, lehnt unsignierte Verteilungen ab.

3.5 SEALED SENDER

```
Datei: src/Rede.Core/Crypto/SealedSender.cs (71 Zeilen)
```

Sealed Sender verbirgt die Absenderidentität vor dem Server. Nur der Empfänger kann den Absender aus dem verschlüsselten Umschlag extrahieren.

Mechanismus:

```
1. Einmal-Ephemeral-Schlüsselpaar generieren
2. Domain-Tag "SEALED_SENDER_V1:" voranstellen (Cross-Protocol-Schutz)
3. Payload = nacl.box(domainTag || innerPayload, nonce, ephSecret, recipientPub)
4. Server sieht nur: {ephemeralKey, nonce, ciphertext} + Empfänger
```

Einschränkungen:

- Nur für 1:1-Nachrichten mit etablierter Ratchet-Sitzung
- Nicht für Erstkontakt (X3DH) oder Gruppen (Server braucht Mitgliederlisten zum Fan-out)

Wire-Grösse: Die Sealed-Cipher base64-expandiert in der Praxis auf ~29.500 chars bei vollständig gepaddeter DR-Nachricht (16382 Bucket + secretbox + JSON-Wrapper + nacl.box). Server-Cap: 32768. Beim Ändern von `MaxAvatarBase64OnWire` (`ChatService.cs:210`, derzeit 12 KiB) oder der Padding-Buckets immer auch den Server-Cap prüfen.

Stille Decode-Pfade in `HandleSealedMessage` (`ChatService.cs:509`): Acht mögliche `return`-Stellen ohne Fehlermeldung - z.B. wenn `Profile.Contacts` den `from`-User nicht enthält, oder wenn `_nonceTracker.Check(nonce)` einen Replay erkennt. Bei "Nachricht kommt einfach nicht an"-Symptomen erst hier nachschaün, bevor man Federation-Routing verdächtigt.

3.6 NACHRICHTENPADDING

```
Datei: src/Rede.Core/Crypto/MessagePadding.cs (57 Zeilen)
```

Alle Nachrichten werden auf feste Grössen aufgefüllt, um Verkehrsanalyse zu erschweren.

Buckets: 256, 1024, 4096, 16384 Bytes

Format:

```
| 2 Byte Länge (Big-Endian) | Nachrichteninhalt | Zufallsauffüllung |
```

Maximale Nachrichtengrösse: 16.382 Bytes (16.384 - 2 Byte Prefix). Wer mehr senden möchte (z.B. Anhänge), muss den Blob-Pfad nutzen.

3.7 HKDF-SHA256

```
Datei: src/Rede.Core/Crypto/Hkdf.cs (95 Zeilen)
```

Implementation von RFC 5869. Verwendet für X3DH und Double Ratchet.

- `Extract(salt, ikm)`: HMAC-SHA256(salt, ikm)
- `Expand(prk, info, length)`: Iteratives HMAC mit Byte-Counter
- `DeriveKey(ikm, salt, info, length)`: Extract + Expand
- `X3dhIdentitySalt(pubA, pubB)`: SHA256("RedeX3DHSalt" || sorted(pubA, pubB))

Alle HMAC-Zwischenergebnisse (PRK, T-Blöcke) werden nach Gebrauch genullt.

3.8 PROFILVERSCHLÜSSELUNG (AT-REST)

```
Datei: src/Rede.Core/Storage/ProfileStore.cs (740 Zeilen)
```

Profile werden auf der Festplatte unter `~/.rede/{sha256(userId)}.enc` verschlüsselt gespeichert.

Verfahren:

```
key = scrypt(passphrase, salt=sha256(userId), N=2^20, r=8, p=1, dkLen=32)
ciphertext = nacl.secretbox(profile_json_utf8, random_nonce, key)
file = nonce (24 Bytes) || ciphertext
```

Optimierungen:

- Scrypt-Schlüssel wird gecacht (~1 ms statt 0.5-2 s pro Speichervorgang)
- Speichervorgänge sind debounced (500 ms Koaleszenz via `CancellationTokenSource`)
- JSON-Serialisierung über Source-Generated Serializer (`ProfileJsonContext`)
- UTF-8-Bytes direkt serialisiert (kein String-Zwischenschritt)
- **`FlushAsync()` muss vor Programmende aufgerufen werden** - sonst geht der letzte Debounce-Schreibvorgang verloren

Atomare Schreibvorgänge: Schreibt nach `~/.rede/{sha256(userId)}.enc.tmp`, fsync, rename. Auf Linux/macOS zusätzlich `O_EXCL`-basierter Lock + Retry-Loop (Audit-Fix H4), damit zwei laufende Instanzen sich nicht gegenseitig die Datei zerschiessen.

3.9 SCHLÜSSELZEROISIERUNG

```
Dateien: CryptoService.cs, SecureMemory.cs, Models.cs
```

Alle geheimen Schlüssel werden als `byte[]` gehalten und nach Gebrauch aktiv genullt:

Drei Schichten:

1. **`CryptographicOperations.ZeroMemory()`**: Zeroisiert einzelne Arrays nach Gebrauch.
2. **`Profile.ZeroSecrets()`**: Nullt alle Profilschlüssel auf einmal (SecretKey, SigningSecretKey, SignedPreKey, OTPs, Gruppen-Keys, Place-MetadataKeys). Wird bei Logout / Window-Close aufgerufen.
3. **`SecureMemory.Lock()`**: OS-Level Memory-Locking via P/Invoke:
    - Linux/macOS: `mlock(2)` - verhindert Swap-Auslagerung
    - Windows: `VirtualLock` - gleiche Funktion
    - `GCHandle.Pinned` verhindert GC-Verschiebung
    - Stilles No-Op wenn der Kernel ablehnt (RLIMIT_MEMLOCK).

Passphrase-Schutz (`SecureTextBox.cs`):

- Custom Avalonia-Control, interne mlock'd `byte[4096]`
- UTF-8 direkt in Buffer geschrieben (kein `string`-Zwischenschritt)
- Backspace entfernt UTF-8-Continuation-Bytes korrekt und nullt die Bytes
- `TextInputMethodClient`-Registrierung: IME-System wird informiert, dass das Control Texteingabe akzeptiert (Fix für IBus/Fcitx/Wayland auf Linux)
- Sichtbarkeits-Toggle (Auge-Icon) zum Anzeigen/Verbergen der Passphrase
- Einziges String-Fenster: `TextInputEventArgs.Text` (1-2 Zeichen, Mikrosekunden)

3.10 SRTP (SPRACHANRUFE)

```
Datei: src/Rede.Core/Audio/SrtpCrypto.cs (214 Zeilen)
```

1:1-Anrufe verwenden SRTP gemäss RFC 3711:

Parameter	Wert
Verschlüsselung	ÄS-128-CM (Counter Mode)
Authentifizierung	HMAC-SHA1-80 (truncated)
Master Key	16 Bytes
Master Salt	14 Bytes
Auth Tag	10 Bytes

Schlüsselableitung:

```
cipherKey   = SRTP_KDF(masterKey, masterSalt, label=0x00, 16)
authKey     = SRTP_KDF(masterKey, masterSalt, label=0x01, 20)
sessionSalt = SRTP_KDF(masterKey, masterSalt, label=0x02, 14)
```

Schlüsselaustausch: Die SRTP-Master-Keys werden im `CALL_OFFER`-Payload über die bestehende Double-Ratchet-Sitzung verschlüsselt ausgetauscht. Der Server hat nie Zugang zum Audiostrom - er sieht nur opake binäre Frames im selben WebSocket.

3.11 SFRAME (GRUPPENANRUFE)

Gruppenanrufe über LiveKit SFU verwenden SFrame für Ende-zu-Ende-Verschlüsselung:

```
SFrame-Key = HKDF(Place.MetadataKey || Group.Key,
                  salt="REDE_GCALL_SFRAME_V1:{scope.Key}",
                  info="sframe", 32)
```

Weder der Rede-Server noch die LiveKit-SFU haben Zugang zum SFrame-Schlüssel. Kein Klartextfallback - Anruf bricht ab, wenn E2EE-Setup fehlschlägt.

4. Protokoll (Client-Sicht)

Das Wire-Format ist in der Server-Doku §3 detailliert beschrieben. Hier nur die Aspekte, die für den Client relevant sind.

Was der Client tut:

1. **Senden:** Konstruiere JSON-Payload, sende über `_conn.Send(Msg.Foo, payload)` (`RedeConnection.cs`). Keine clientseitige Signatur des Outer-Envelopes - die Authentifizierung passiert per Auth-Challenge.
2. **Empfangen:** `ReceiveLoop` (`RedeConnection.cs:354-366`) verifiziert die `serverSig` mit dem **gepinnten** Server-Signing-Key, bevor irgendein Service-Handler die Nachricht sieht. Fehlende oder ungültige Signatur -> Nachricht wird verworfen, OnError ausgelöst.
3. **Dispatch:** `_handlers[type](msg)` - jeder Service registriert seine Handler in `_conn.On(Msg.Foo, HandleFoo)`. Unbekannte Types werden silent ignoriert - mit einer Ausnahme: `Msg.Error` fällt seit v2.19.10 in einen generischen `OnError("[Server] ...")`-Banner, wenn kein Service-Handler registriert ist (siehe `RedeConnection.cs:405`).

Nachrichtenumschlag (`MessageEnvelope.cs`): Innerhalb des ratchet-verschlüsselten Klartext-Blobs sendet der Client ein 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 als Legacy-Nachrichten interpretiert (Rückwärtskompatibilität mit v1).

Kontrollnachrichten (`__rede_ctrl`) gehen über denselben verschlüsselten Kanal:

- `{"__rede_ctrl": "profile", ...}` - Profiländerungen (Avatar, Akzentfarbe, MIME-Typ)
- `{"__rede_ctrl": "reaction", "mid": "...", "emoji": "...", "action": "add"}` - Reaktionen
- `{"__rede_ctrl": "edit", "mid": "...", "newText": "..."}` - Nachricht bearbeiten
- `{"__rede_ctrl": "delete", "mid": "..."}` - Nachricht löschen

Profil-Synchronisation: Profiländerungen werden via `BroadcastProfile` an alle Kontakte gesendet. Zusätzlich hält `ChatService._profileSentFingerprint` einen Per-Sitzungs-Cache aus 12-Byte SHA-256-Fingerabdrücken (`accent|avatar|mime`). Beim nächsten ein- oder ausgehenden Chatkontakt vergleicht `EnsureProfileSentTo` den aktüllen mit dem zuletzt gesendeten Fingerabdruck und sendet das Profil erneut, falls es sich geändert hat. Das deckt asymmetrische Adds (das initiale Profil wurde verworfen, weil der Empfänger den Sender noch nicht als Kontakt hatte) und Broadcasts an Offline-Kontakte ab.

`MsgId`-Zuweisung (ACK-FIFO): Outgoing-Sends pushen die lokal erstellte `ChatMessage` auf eine FIFO (`_pendingAck` in jedem Chat-Service). Der Server stempelt `msgId` und antwortet mit `MESSAGE_ACK`/`SEALED_MESSAGE_ACK`; der erste ACK deqüüd den FIFO-Head und schreibt `msgId` in die persistierte Message. Bei Multi-Device-Fan-out (N Devices) gibt es N FIFO-Pushes - der erste stempelt, die restlichen finden `msgId` bereits gesetzt und sind No-Ops. **Niemals nach Inhalt/Timestamp matchen** - konkurrente Edits/Reactions können Messages mit identischem Text/Timestamp produzieren und sich gegenseitig die `msgId` klaün.

5. Client v2 - Desktop (Avalonia)

5.1 PROJEKTSTRUKTUR

```
src/
|-- Rede.Core/                        # Plattformunabhängige Geschäftslogik
|   |-- Audio/
|   |   |-- Audiöngine.cs            # PortAudio + Opus (500 Zeilen)
|   |   |-- RNNoise.cs                # P/Invoke-Wrapper für neuronale Rauschunterdrückung
|   |   |-- SrtpCrypto.cs             # RFC 3711 SRTP (214 Zeilen)
|   |   +-- SrtpSession.cs            # SRTP-Sitzungsmanager
|   |-- Crypto/
|   |   |-- Base64BytesJsonConverter.cs # byte[] <-> Base64-String JSON-Konvertierung
|   |   |-- CryptoService.cs          # Kernkryptografie (303 Zeilen)
|   |   |-- DoubleRatchet.cs          # Signal-Double-Ratchet (431 Zeilen)
|   |   |-- Hkdf.cs                   # HKDF-SHA256 (95 Zeilen)
|   |   |-- MessagePadding.cs         # Festgrössen-Padding (57 Zeilen)
|   |   |-- NonceTracker.cs           # Replay-Schutz
|   |   |-- PQKem.cs                  # ML-KEM-768 Wrapper (BouncyCastle)
|   |   |-- ProfileEncryption.cs      # scrypt + nacl.secretbox (179 Zeilen)
|   |   |-- SealedSender.cs           # Anonymer Absender (71 Zeilen)
|   |   |-- SecureMemory.cs           # OS-Level Memory Lock (98 Zeilen)
|   |   |-- SenderKeys.cs             # Gruppen-PFS (155 Zeilen)
|   |   +-- X3dh.cs                   # X3DH/PQXDH-Schlüsseltausch (~270 Zeilen)
|   |-- Networking/
|   |   |-- ProxySettings.cs          # I2P/Tor/Direct Konfiguration
|   |   +-- RedeConnection.cs         # WebSocket + TOFU + Reconnect (479 Zeilen)
|   |-- Protocol/
|   |   |-- MessageTypes.cs           # Typenregister (127 Zeilen)
|   |   +-- ProtocolSerializer.cs     # JSON-Serialisierung
|   |-- Services/
|   |   |-- AuthService.cs            # Registrierung + Login (340 Zeilen)
|   |   |-- AutostartService.cs       # OS-Autostart (.desktop / Registry)
|   |   |-- BlobService.cs            # Dateianhang-Upload/Download (208 Zeilen)
|   |   |-- CallService.cs            # 1:1 Anruf-State-Machine (~660 Zeilen)
|   |   |-- ChatService.cs            # Nachrichten senden/empfangen (~830 Zeilen)
|   |   |-- ContactService.cs         # Kontaktverwaltung (228 Zeilen)
|   |   |-- DeviceService.cs          # Geräteverknüpfung
|   |   |-- DiscordImportService.cs   # Discord-Server-Import (510 Zeilen)
|   |   |-- GroupCallService.cs       # LiveKit-Gruppenanrufe (279 Zeilen)
|   |   |-- GroupService.cs           # Gruppenverwaltung (501 Zeilen)
|   |   |-- MessageEnvelope.cs        # Nachrichtenumschlag-Kodierung (168 Zeilen)
|   |   |-- NotificationService.cs    # Cross-Platform-Benachrichtigungen (252 Zeilen)
|   |   |-- PlaceService.cs           # Place-Verwaltung (1872 Zeilen)
|   |   +-- UpdateService.cs          # Auto-Update + SHA256-Verifikation (565 Zeilen)
|   +-- Storage/
|       |-- Models.cs                  # Alle Datenmodelle (570 Zeilen)
|       +-- ProfileStore.cs           # Verschlüsselte Profilespeicherung (740 Zeilen)
|
+-- Rede.Desktop/                     # Avalonia 11 UI-Projekt
    |-- App.axaml / .cs               # Application-Einstieg
    |-- Program.cs                    # Main-Methode
    |-- MainWindow.axaml / .cs        # Hauptfenster (2749 Zeilen Code-Behind!)
    |-- Controls/
    |   |-- MarkdownTextBlock.cs      # Inline-Markdown-Renderer
    |   +-- SecureTextBox.cs          # Passphrase-Eingabe mit mlock (309 Zeilen)
    |-- Converters/
    |   +-- Converters.cs             # SidebarWidth, StatusColor, CollapseIcon
    |-- Themes/
    |   |-- Colors.axaml              # Farbpalette
    |   |-- Icons.axaml               # Lucid Pixel-Art Icons (579 Zeilen)
    |   |-- RedeTheme.axaml           # Globale Stile
    |   |-- ThemeService.cs           # Live Theme-Switching
    |   +-- Typography.axaml          # Schriftgrössen
```

```
    |-- ViewModels/
    |   |-- CallViewModel.cs          # Anruf-UI-State
    |   |-- LoginViewModel.cs         # Login/Register-State (270 Zeilen)
    |   |-- MainViewModel.cs          # Sidebar, Chat, Nachrichten (576 Zeilen)
    |   |-- PlaceSettingsViewModel.cs  # Place-Einstellungen (339 Zeilen)
    |   |-- SettingsViewModel.cs      # Einstellungen (338 Zeilen)
    |   +-- ViewModelBase.cs          # CommunityToolkit.Mvvm Basis
    +-- Views/
        |-- BootView.axaml / .cs      # Terminal-Style Boot-Animation
        |-- CallView.axaml / .cs      # 1:1 Anruf-Overlay
        |-- GroupCallWindow.axaml / .cs # WebView-basiertes Gruppenanruf-Fenster
        |-- LoginView.axaml / .cs     # Login/Register-Formular
        |-- MainView.axaml / .cs      # Sidebar + Chat-Layout
        |-- PlaceSettingsView.axaml / .cs # Discord-Style Place-Settings
        +-- SettingsView.axaml / .cs  # Einstellungen-Panel
```

5.2 REDE.CORE - GESCHÄFTSLOGIK

`Rede.Core` ist eine reine .NET 8 Klassenbibliothek ohne UI-Abhängigkeiten. Sie enthält die gesamte Kryptografie, Netzwerkkommunikation, Zustandsverwaltung und Geschäftslogik.

NuGet-Abhängigkeiten:

Paket	Version	Verwendung
Sodium.Core	1.4.0	libsodium-Bindings (NaCl)
Concentus	2.2.2	Opus-Codec (managed)
PortAudioSharp2	1.0.0	Cross-Platform-Audio-I/O
BouncyCastle.Cryptography	2.6.2	ML-KEM-768 (PQXDH)
Microsoft.Win32.Registry	5.0.0	Windows-Registry (Autostart)

Native Abhängigkeiten (optional, nicht gebundelt):

- `librnnoise.so` / `rnnoise.dll`: Neuronale Rauschunterdrückung (xiph.org, BSD-3). Nicht im Binary enthalten - per One-Click-Install in Settings oder `scripts/install-rnnoise.sh` / `.ps1` in `~/.rede/libs/` installierbar. Custom `NativeLibrary.SetDllImportResolver` sucht beim Start in `~/.rede/libs/` und neben der Exe.

5.3 REDE.DESKTOP - UI

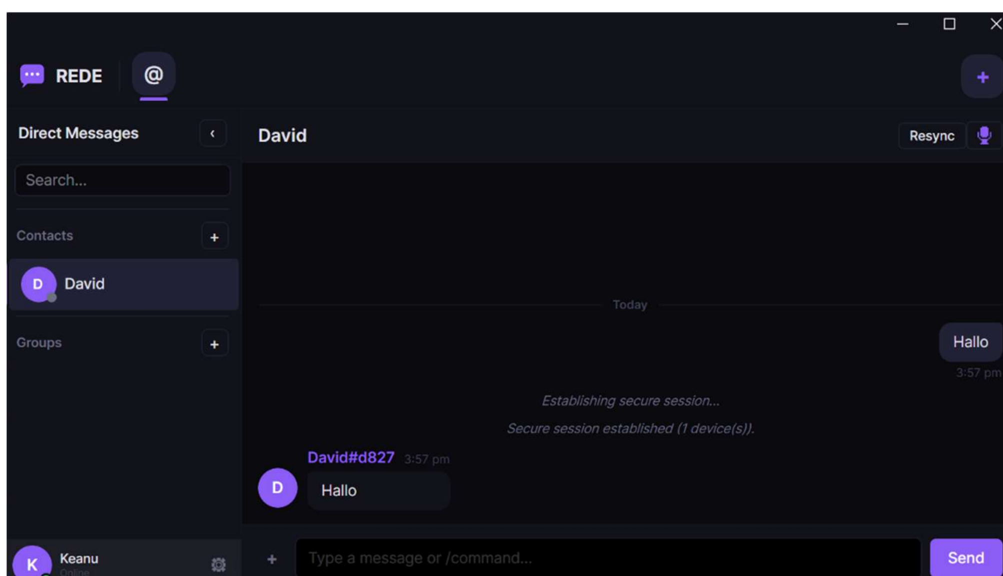
`Rede.Desktop` ist das Avalonia 11 UI-Projekt. Es produziert ein einzelnes selbstenthaltendes Executable.

NuGet-Abhängigkeiten:

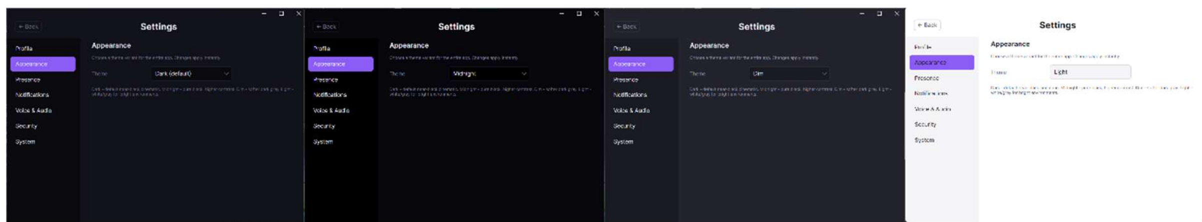
Paket	Version	Verwendung
Avalonia	11.3.12	UI-Framework
Avalonia.Desktop	11.3.12	Desktop-Backend
Avalonia.Themes.Flünt	11.3.12	Basis-Theme
Avalonia.Fonts.Inter	11.3.12	Inter-Schriftart
CommunityToolkit.Mvvm	8.4.1	MVVM-Infrastruktur
Microsoft.Extensions.DI	10.0.5	Dependency Injection
WebView.Avalonia	11.0.0.1	Gruppenanrufe (LiveKit)

Design:

- Chromloses Fenster: Titelleiste integriert in App-Inhalt, Mindestgrösse 800x550
- Hintergrund: Near-Black (`#0a0a0f`)
- Akzentfarben: Violett (`#8b5cf6`), Teal (`#2dd4bf`)

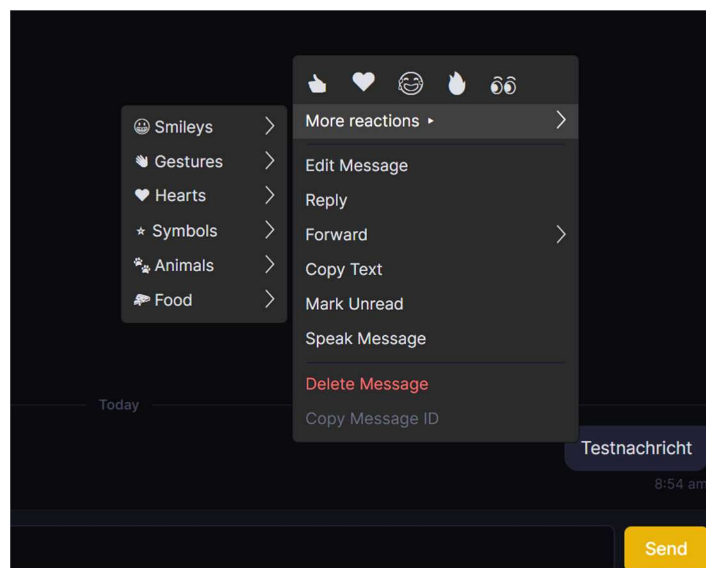


- Inter-Schrift, 15 px Body
- Lucid Pixel-Art Icons (CC0, themed mit App-Akzentfarben)
- 4 Theme-Varianten: Dark, Midnight, Dim, Light (live switchbar)



- Nachrichten-Bubbles: dynamische Breite (füllt Chat-Bereich minus 80 px Margin), passt sich bei Fenster-Resize an
- Inline-Markdown in Nachrichten: `*fett*`, `_kursiv_`, `` `code` ``, ```` ```codeblock``` ````, `~~durchgestrichen~~`
- "Newest Messages"-Button: Accent-farbener Pill-Button am unteren Chat-Rand, erscheint beim Hochscrollen, Klick scrollt nach unten. Auto-Scroll nur wenn User am Ende ist.

Nachrichten-Kontextmenü (Rechtsklick) im Discord-Stil: Add Reaction (Emoji-Schnellauswahl, 8 Emojis, Toggle), Edit Message (eigene), Reply, Forward (Submenü mit Kontakten/Gruppen), Copy Text / Copy Selection (bevorzugt selektierten Text), Pin Message (Places), Mark Unread, Speak Message (TTS via spd-say/say/SAPI), Delete Message (rot), Copy Message ID (grau).



- Profil-Broadcast: Avatar und Akzentfarbe werden automatisch beim Login an alle Kontakte und beim Hinzufügen an den neün Kontakt gesendet.

5.4 DIENSTE (SERVICES)

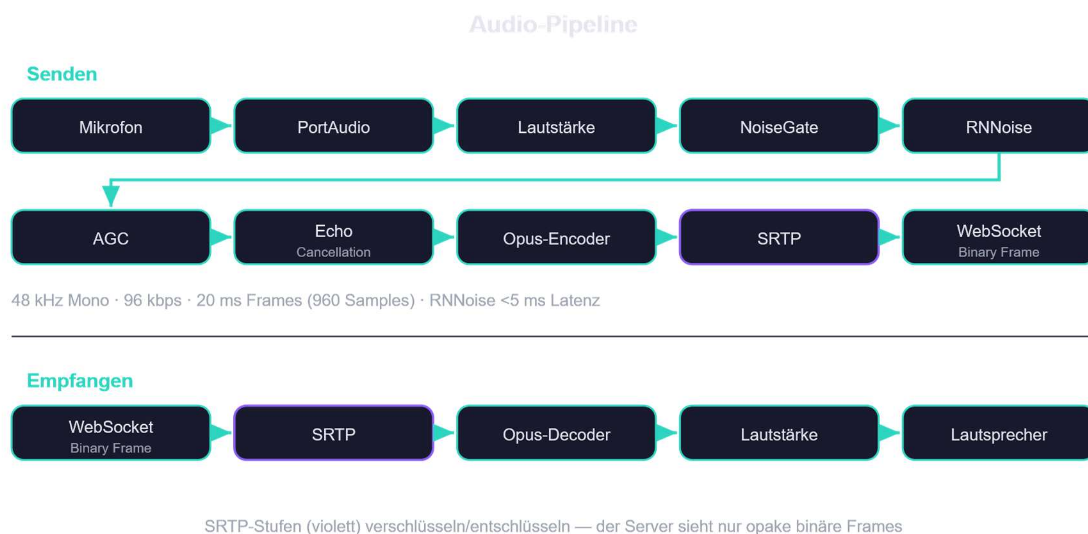
Jeder Service kümmert sich um einen abgegrenzten Verantwortungsbereich und kommuniziert über Events:

Service	Verantwortung
AuthService	Registrierung, Login, Auth-Challenge, Geräteverknüpfung. `UploadPreKeysIfNeeded(prekeyCount)` lauft im `AUTH_OK`-Handler, lade nach wenn `prekeyCount <= 5`
ChatService	1:1-Nachrichten senden/empfangen, X3DH-Initiierung, Nachrichtenverlauf, Profil-Broadcast, Multi-Device-Fan-out + Legacy-Fallback, ACK-FIFO
ContactService	Kontakte hinzufügen/entfernen/bestätigen, Fingerabdrücke, Profilverteilung
GroupService	Gruppengründung, Einladung, Kick, Rekey, Gruppenachrichten
PlaceService	Place-CRUD, Kanalverwaltung, Rollen, Bans, Emotes, Metadaten-E2EE (1872 Zeilen)
CallService	1:1-Anruf-State-Machine, SRTP-Pipeline, Double-Ratchet-Schlüsselaustausch, Legacy-Fallback (v2.19.11)
GroupCallService	LiveKit-Token-Anfrage, SFrame-Schlüssableitung, Anrufankündigung
BlobService	Verschlüsselter Blob-Upload/Download, LRU-Cache + Disk-Cache (`~/.rede/blobs/{sha256(userId)}/...bin`)
DeviceService	Multi-Device-Verknüpfung und -Verwaltung
UpdateService	Auto-Update via Git oder GitHub Releases API, Ed25519-Signatur + SHA256, `pkexec`-Elevation auf Linux
NotificationService	OS-Benachrichtigungen (notify-send, PowerShell, osascript)
DiscordImportService	Discord-Server-Import via Bot-API
AutostartService	OS-Autostart (.desktop-Datei / Registry)

Wichtige Service-Pattern:

- **Events statt direkter Aufrufe** zwischen Services - z.B. `ChatService.OnMessageReceived` wird vom UI / MainWindow abonniert.
- **`OnSystemMessage`-Banner**: Jeder Service hat einen `event Action<string> OnSystemMessage`; MainWindow Z. 928 routet zu `_mainVm.AddSystemMessage`. Fehler, Statushinweise, "Establishing secure session..." landen hier.
- **`OnError`-Toast**: Für Fehler, die nicht in einen spezifischen Chat gehören (z.B. Server-ERROR-Banner, TOFU-Warnungen).

5.5 AUDIO-PIPELINE



Audioengine.cs (500 Zeilen):

- PortAudioSharp2 für plattformübergreifende Audiogerätezugriff
- Concentus Opus: 48 kHz, Mono, 96 kbps Bitrate (über Discord Standard, 64 kbps), Komplexität 8
- 20 ms Frames (960 Samples)
- Input-Callback: Volume Adjustment -> Noise Gate -> RNNoise -> AGC -> Echo Cancellation -> Opus Encode
- Monitor-Modus: Nur Mic-Level-Metering ohne Encoding (für Settings)

RNNoise:

- Neuronale Rauschunterdrückung (xiph.org)
- P/Invoke-Wrapper mit graceful Fallback wenn Native-Lib fehlt
- Custom `NativeLibrary.SetDllImportResolver` sucht in `~/.rede/libs/` und neben der Exe
- One-Click-Install über Settings UI (Download von GitHub Releases, `TryReload()` hot-load)
- 960 Samples als zwei 480-Sample-RNNoise-Frames verarbeitet
- <5 ms Latenz

5.6 THEMEN UND STYLING

Theme-Varianten (live switchbar ohne Neustart):

Variante	Hintergrund	Primär
Dark	`#0a0a0f`	Violett
Midnight	`#0f0f23`	Blau
Dim	`#1a1a2e`	Gedämpft
Light	`#f5f5f5`	Dunkelviolett

`ThemeService.cs` wendet Farbpaletten als dynamische Ressourcen an. `Profile.ThemeVariant` und `Profile.AccentColor` persistieren die Auswahl.

12 Akzentfarb-Presets: Violett, Teal, Blau, Grün, Gelb, Orange, Rot, Pink, Rose, Indigo, Cyan, Smaragd.

6. Client v1 - Terminal (Node.js)

Der v1-Client bleibt als vollwertiger Terminal-Client neben v2 funktionsfähig.

```
PowerShell 7 (x64)
rede :: Keanu (Keanu#bf57) [0016f31c] | E2EE + PFS
commands:
  /add <id#xxxx>       add contact
  /confirm <id#xxxx>   accept key change
  /reset <id#xxxx>     reset ratchet session
  /fingerprint [user]  show fingerprint
  /group <name>        create group
  /ginvite <grp> <usr> invite to group
  /kick <grp> <usr>    remove from group
  /rekey <group>       rotate group key
  /ttl <seconds>       self-destruct (0=off)
  /contacts            list contacts
  /groups              list groups
  /key                 show your public key
  /help                show this
  /quit                exit

tab = switch focus | ctrl+c = quit
> Device linked successfully! Device ID: 0016f31c
> Pre-keys uploaded (20 OTPs).
> _
```

Datei	Zeilen	Funktion
`client/index.js`	1336	TUI-Einstieg, Split-Pane-Layout, alle Nachrichtenhandler
`client/cli.js`	1268	CLI-Modus (parallele Implementierung - Sicherheitsfixes MÜSSEN in beide!)
`client/crypto.js`	1017	X3DH, Double Ratchet, Sender Keys, Sealed Sender, Padding
`client/store.js`	376	Verschlüsselte Profilspeicherung (scrypt + nacl.secretbox, atomare Schreibvorgänge mit O_EXCL-Lock)
`client/network.js`	319	WebSocket-Client, Server-Signaturverifikation, Sealed Sender
`client/ui.js`	265	Reine ANSI-Terminal-UI (keine Abhängigkeiten)
`client/boot.js`	203	Startup, Profilanlage/-erkennung

Abhängigkeiten (Node.js): tweetnacl 1.0.3, tweetnacl-util 0.15.1, ws 8.19.0, socks-proxy-agent 8.0.5.

Wichtig: `index.js` und `cli.js` sind parallele Implementierungen. Jeder Sicherheitsfix muss in BEIDE Dateien eingepflegt werden.

PQXDH: v1 implementiert PQXDH nicht (kein BouncyCastle-Äquivalent in der Node-Toolchain). Bei Kommunikation mit einem v2-Peer fallen beide Seiten transparent auf klassisches X3DH zurück.

7. Datenmodell (im Client)

```
Datei: src/Rede.Core/Storage/Models.cs (570 Zeilen)
```

7.1 PROFIL

```
Profile {
    UserId, DisplayName, DeviceId,
    PublicKey, SecretKey,              // X25519 (byte[])
    SigningKey, SigningSecretKey,      // Ed25519 (byte[])
    Contacts: Dict<string, Contact>,
    Groups: Dict<string, Group>,
    Places: Dict<string, Place>,
    ChatHistory: Dict<string, List<ChatMessage>>,   // pro Contact/Group/Place:Channel
    SignedPreKey, SignedPreKeySig,     // X3DH
    OneTimePreKeys: List<OneTimePreKey>,
    PreviousSignedPreKeys: List<ArchivedSignedPreKey>, // max 5, für Rotation-Decrypt
    PqSignedPreKey, PqSignedPreKeySig,          // PQXDH (optional)
    PqOneTimePreKeys: List<OneTimePreKey>,
    PreviousPqSignedPreKeys: List<ArchivedSignedPreKey>,
    RatchetStates: Dict<string, JsonElement>,  // pro Contact:Device (oder legacy: nur Contact)
    SenderKeys: Dict<string, JsonElement>,     // pro Group/Place:Channel
    ServerSigningKey: byte[],          // TOFU-gepinnt
    OwnDevices: Dict<string, DeviceKeys>,
    SeenNonces: Dict<string, long>,    // Replay-Schutz (persistiert)
    DeliveryToken: string,             // 24h-Token für Sealed-Sender
    PendingKeyChange: ...,             // ungesehene Key-Change-Warnungen
    // Audio
    InputDeviceName, OutputDeviceName, InputVolume, OutputVolume,
    NoiseGateThreshold, NoiseSuppression, AutoInputSensitivity, AutoGainControl, EchoCancellation,
    // Theme, Status, Notifications, Tray
    AccentColor, AvatarData, AvatarMimeType, ThemeVariant, LastServerName,
    Status, CustomStatus,
    NotificationsEnabled, NotificationShowContent, NotificationSoundEnabled,
    MinimizeToTray, Autostart, StartMinimized,
    // Call
    DefaultCallMode, AllowFastCalls
}
```

Gespeichert verschlüsselt unter: `~/.rede/{SHA256(userId)}.enc`.

`ZeroSecrets()` nullt SecretKey, SigningSecretKey, SignedPreKey-Secret, OTP-Secrets, Group-Keys und Place-MetadataKeys auf einmal - wird bei Logout/Window-Close gerufen.

7.2 KONTAKT

```
Contact {
    PublicKey, SigningKey,              // byte[]
    Devices: Dict<string, DeviceKeys>, // Pro Gerät eigene Schlüssel
    DisplayName, Alias,
    AccentColor, AvatarData, AvatarMimeType, // Empfangenes Profil
    Status, CustomStatus                // Empfangener Präsenzstatus
}
```

Stolperfalle Devices-Map: `HandlePrekeyBundle` (`ChatService.cs:890`) aktualisiert `Contact.Devices` *nicht*, obwohl die PREKEY_BUNDLE-Antwort die per-device-Keys mitliefert. Folge: Bei Altkontakten, deren Devices-Map bei `AddContactAsync` leer war, bleibt sie auch leer - der Send- und Call-Pfad fallen dann auf den Legacy-Ratchet-State (`Profile.RatchetStates[contactId]` ohne `:deviceId`) zurück. Ein sauberer Fix würde die Devices-Map im `HandlePrekeyBundle`-Pfad nachziehen.

7.3 GRUPPE

```
Group {
    Name: string,
    Key: byte[],                        // 32 Byte symmetrischer Schlüssel
    Members: List<string>               // UserId-Liste
}
```

7.4 PLACE

```
Place {
    Name, MetadataKey: byte[],        // 32 Byte für E2EE-Metadaten
    Channels: Dict<string, PlaceChannel>,
    Members: List<string>,
    Roles: Dict<string, PlaceRole>,           // Legacy 3-Tier (Owner/Admin/Member)
    CustomRoles: Dict<string, CustomRole>,    // Unbegrenzte Custom-Rollen
    MemberRoles: Dict<string, List<string>>,  // userId -> roleIds
    CreatorId: string,
    // Profil
    AccentColor, IconData, IconMimeType,
    OwnerColor, AdminColor, MemberColor,      // Rollenfarben
    // Features
    Emotes: Dict<string, PlaceEmote>,         // max 50, max 64 KB/Emote
    Bans: Dict<string, PlaceBan>,
    Pins: Dict<string, List<PlacePin>>,       // channelId -> Pins (max 50/Kanal)
    Nicknames: Dict<string, string>,          // userId -> Anzeigename (per-Place override)
    Categories: List<string>                  // Geordnete Kategorien
}

PlaceChannel {
    Name, Topic (max 200 chars),
    Category (optional), Position (Reihenfolge in Kategorie),
    CreatedAt,
    PermissionOverrides: Dict<string, ChannelPermOverride>  // pro Rolle Allow/Deny-Bitmask
}
```

Alle Metadaten (Name, Topic, Emotes, Bans, Pins, Nicknames, CustomRoles) sind clientseitig mit `MetadataKey` (nacl.secretbox) verschlüsselt - der Server sieht nur opake IDs und Roster.

7.5 CHATNACHRICHT

```
ChatMessage {
    From, Text, Ts (Unix ms),
    Ttl (Tage, 0=permanent),
    MsgId (server-generiert),
    // Antworten
    ReplyToMsgId, ReplyToPreview (max 100 Zeichen), ReplyToAuthor,
    // Reaktionen
    Reactions: Dict<string, List<string>>,  // Emoji -> UserIds
    // Bearbeitung/Löschung
    EditedAt (Unix ms, null falls nie), IsDeleted,
    // Anhänge
    Attachments: List<AttachmentInfo>       // BlobId + Schlüssel + Nonce + Name
}

AttachmentInfo {
    BlobId, Key, Nonce, Name, MimeType (optional), Size
}
```

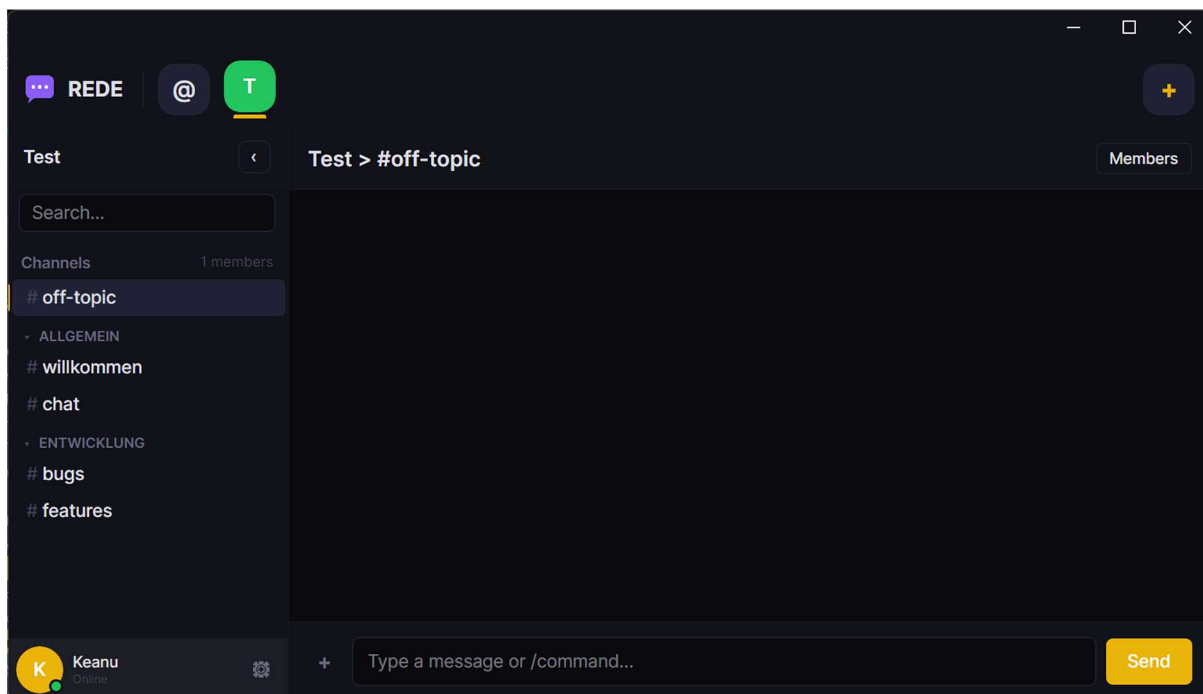
Schlüssel/Nonce der Anhänge leben nur im (passphrase-verschlüsselten) Chatverlauf - der Server speichert nur Chiffretext der Blobs. Damit Bilder/Dateien nach Server-TTL (30 Tage) oder Cache-Eviction noch verfügbar sind, persistiert `BlobService` den Chiffretext lokal unter `~/.rede/blobs/{sha256(userId)}/{blobId}.bin`. Beim `FetchAsync` wird zürst der In-Memory-LRU geprüft, dann disk, dann Server.

8. Places - Discord-ähnliche Server

8.1 KANALSTRUKTUR

```
Place

    +-- #offtopic
|-- Kategorie "Allgemein"
|   |-- #willkommen   (Position 0)
|   +-- #chat         (Position 1)
|-- Kategorie "Entwicklung"
|   |-- #bugs         (Position 0)
|   +-- #features     (Position 1)
+-- (ohne Kategorie)
```



8.2 ROLLENSYSTEM

Legacy (3-Tier): Member < Admin < Owner.

Custom Roles (ab Phase U.6):

```
CustomRole {
    Id, Name, Color,
    Position: int,          // Höher = mehr Macht
    Permissions: long       // Bitfeld
}
```

`InitializeDefaultRoles()` migriert vom 3-Tier-System:

- `@everyone` (Position 0, SendMessages)
- `Admin` (Position 1, Administrator)
- `Owner` (Position 2, Administrator)

8.3 BERECHTIGUNGEN

```
[Flags] enum PlacePermission : long {
    SendMessages    = 1 << 0,
    ManageMessages  = 1 << 1,  // Fremde Nachrichten löschen
    ManageChannels  = 1 << 2,
    ManageRoles     = 1 << 3,
    KickMembers     = 1 << 4,
    BanMembers      = 1 << 5,
    ManageEmotes    = 1 << 6,
    ManagePlace     = 1 << 7,  // Name/Icon/Akzent
    Administrator   = 1 << 8,  // Alle Berechtigungen
    ViewAuditLog    = 1 << 9
}
```

Berechtigungsauflösung:

```
Effektive Berechtigungen = Basis-Rollen-Permissions
                         + Kanal-Overrides (Allow)
                         - Kanal-Overrides (Deny)
```

`HasCustomPermission()` prüft die höherrangigste Rolle des Benutzers.

8.4 METADATEN-VERSCHLÜSSELUNG

Alle Place-Metadaten werden mit `MetadataKey` (nacl.secretbox) verschlüsselt:

- Kanalnamen und -themen
- Kategorien
- Emotes (Name + Bilddaten)
- Bans (UserId + Grund)
- Pins
- Nicknames
- Rollenfarben
- Custom Roles und Zuweisungen
- Place-Profil (Icon, Akzentfarbe)

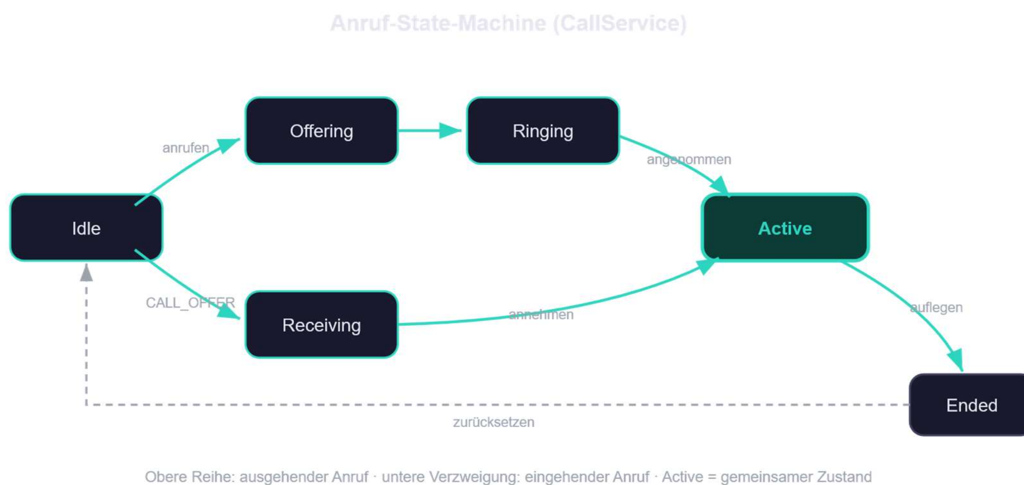
Der Server sieht nur opake Kanal-IDs und Mitgliederlisten. Er hat keinen Zugang zu Inhalten oder Strukturinformationen.

9. Sprachanrufe

9.1 1:1 ANRUFE (SRTP)

```
Datei: src/Rede.Core/Services/CallService.cs (~660 Zeilen)
```

State Machine:



Obere Reihe: ausgehender Anruf · untere Verzweigung: eingehender Anruf · Active = gemeinsamer Zustand

Schlüsselfluss:

```
1. Initiator generiert SRTP Master Key + Salt
2. Verschlüsselt mit bestehendem Double-Ratchet-Schlüssel
3. Im CALL_OFFER-Payload gesendet
4. Empfänger entschlüsselt und leitet Session Keys ab
5. Audio läuft als binäre WebSocket-Frames über den Server
```

Multi-Device-Fan-out + Legacy-Fallback: `srtpParams.perDevice` ist ein Array, in dem der Anrufer den SRTP-Schlüssel **einmal pro Empfänger-Device** verschlüsselt - jedes Element trägt `encrypted`, `nonce`, `header` und `toDeviceId`. Der Server fächert die `CALL_OFFER` an alle Devices des Empfängers; jedes Device sucht im `perDevice`-Array den Eintrag mit seinem eigenen `toDeviceId` und verwirft die Nachricht stumm, wenn keiner passt (sonst würde ein automatischer `crypto_error`-Reject von einem nicht adressierten Device den echten `CALL_ANSWER` überholen).

Zusätzlich existiert ein **Legacy-Pfad** (ab v2.19.11) für Kontakte, deren `Contact.Devices` clientseitig leer ist (Altkontakte, USER_LOOKUP_OK ohne `devices`-Feld, etc.): `CallService.InitiateCall` fällt dann auf den No-Device-Ratchet-State zurück (`_store.LoadRatchetState(profile, target, null)`) und sendet `srtpParams` direkt auf der obersten Ebene **ohne** `perDevice`-Array. `HandleCallOffer` hat dafür einen eigenen "legacy single-device"-Zweig, der diese Form akzeptiert. Ohne diesen Fallback hat ein Kontakt, mit dem ChatService längst per Legacy-Path Nachrichten austauschte, beim Anrufen still mit `No secure session.` abgewiesen - denn `ChatService.SendMessageAsync` hat den Legacy-Fallback schon länger, `CallService.InitiateCall` aber nicht.

Audio-Parameter:

- Opus 96 kbps (über Discord Standard, 64 kbps)
- 48 kHz Mono, 20 ms Frames
- RNNoise neuronale Rauschunterdrückung
- Auto Gain Control, Echo Cancellation
- SRTP: ÄS-128-CM + HMAC-SHA1-80

Transport-Awareness:

- `CallMode` leitet sich vom `RedeConnection.Transport` ab (Direct/I2P/Tor)
- NICHT einstellbar - abhängig vom aktiven Transport
- Server relayed binäre Frames zwischen den Peers; bei Cross-Node-Federation laufen sie per Base64-wrapped Pub/Sub durch Redis (s. Server-Doku §4.3).

9.2 GRUPPENANRUFE (LIVEKIT + SFRAME)

```
Datei: src/Rede.Core/Services/GroupCallService.cs (279 Zeilen)
```

Architektur:

```
Client A ---LiveKit---> LiveKit SFU ---LiveKit---> Client B
   |                       |                          |
   +--SFrame E2EE (Schlüssel nie beim SFU/Server)----+
```

Ablauf:

1. Client sendet `GCALL_REQÜST_TOKEN` mit Scope (Place/Group + Channel)
2. Server generiert HS256 JWT (6 h TTL) mit HMAC-abgeleitetem opaken Raumnamen
3. Client leitet SFrame-Schlüssel lokal ab:sframeKey = HKDF(metadataKey, "REDE_GCALL_SFRAME_V1:{scopeKey}", "sframe", 32)
4. LiveKit WebView-Host (`Assets/gcall/index.html`) mit livekit-client 2.18.1 + E2EE-Worker
5. `ExternalE2EEKeyProvider` erhält den Schlüssel - kein Klartext-Fallback
6. `GCALL_ANNOUNCE` benachrichtigt andere Mitglieder

Limits:

- Max 25 Teilnehmer pro Anruf
- Max 200 gleichzeitige Anrufe
- 1080p60 Video-Ziel
- Opus Audio
- Rate Limit: 10 Token-Anfragen/Minute

Pseudonym-Unlinkability:

```
gcallPseudonym = HMAC-SHA256(userId, callId)
```

Der `callId` wird eingemischt, sodass derselbe Benutzer in verschiedenen Anrufen verschiedene Pseudonyme hat. LiveKit-Operator kann die Pseudonyme nicht auf User-IDs zurückmappen.

10. Transport und Netzwerk

10.1 TRANSPORTMODI

```
Datei: src/Rede.Core/Networking/RedeConnection.cs (~480 Zeilen)
```

Transport	Protokoll	Latenz	IP verborgen
Direct WSS	`wss://` via nginx	~50-100 ms	Nein
I2P	`ws://` über i2pd SOCKS5	~500-2000 ms	Ja
Tor	`ws://` über Tor SOCKS5	~300-1000 ms	Ja

Konfiguration über .env:

```
REDE_SERVER=ws://address.i2p
REDE_TRANSPORT=i2p
REDE_I2P_PROXY=socks5h://127.0.0.1:4447
REDE_TOR_PROXY=socks5h://127.0.0.1:9050
```

Nachrichten sind IMMER E2EE, unabhängig vom Transport. Andere Benutzer sehen nie die IP.

Empfangs-Dispatch (`RedeConnection.cs:354-409`):

```
ReceiveLoop:
  raw = await ws.ReceiveAsync()
  msg = JsonNode.Parse(raw)
  TOFU-Pin-Check (Server-Signing-Key)        // s. §10.3
  VerifyServerSignature(raw, ServerSigningKey)
  type = msg["type"]
  if (type == "qüü_position") OnQüüPosition?.Invoke(...)
  elif (type == "qüü_admit")  OnQüüAdmit?.Invoke()
  elif (type == "error" && !_handlers.ContainsKey("error"))
                              OnError?.Invoke($"[Server] {error}")    // v2.19.10
  elif (_handlers.TryGetValü(type, handler))
                              handler(msg)                              // Service-Handler
  // sonst silent drop
```

10.2 TOFU-ZERTIFIKATSPINNING

Trust-On-First-Use: Das TLS-Zertifikat des Servers wird beim ersten Kontakt gepinnt und bei jedem Folgekontakt verifiziert.

- Pins gespeichert in `~/.rede/.cert_pin` (Windows: `%USERPROFILE%\.rede\.cert_pin`), JSON: `server_url -> SHA256-Fingerprint`
- Unix: Dateiberechtigungen auf 600 gesetzt
- Implementierung: `RedeConnection.TofuValidation` (`src/Rede.Core/Networking/RedeConnection.cs:149`)

Validierungslogik (ab v2.19.1-beta):

Fall	\`SslPolicyErrors\`	Pin-Vergleich	Verhalten
Erster Kontakt	egal	kein Pin vorhanden	akzeptieren, Fingerprint pinnen, TOFU-Hinweis ans UI
Wiederholter Kontakt	egal	Pin == Fingerprint	akzeptieren (silent)
Cert-Rotation (z.B. Let's-Encrypt-Renewal)	`None` (CA-validiert)	Pin != Fingerprint	akzeptieren, **silent Re-Pin** auf neün Fingerprint
Echter Drift / MITM	!= `None`	Pin != Fingerprint	**blockieren**, `[SECURITY]`-Warnung
Klartext-WS ohne Cert	-	-	nur akzeptieren, wenn URL nicht `wss://` ist (z.B. I2P/Tor-Tunnel auf `ws://localhost`)

Hintergrund Re-Pin: Let's-Encrypt-Zertifikate werden alle ~60 Tage automatisch erneürt. Vor v2.19.1 hat jedes Renewal alle bestehenden Clients ausgesperrt - die Pin-Mismatch-Exception kam vom .NET-Stack als nichtssagendes *"Unable to connect to the remote server"* zurück, die `[SECURITY]`-Meldung landete nur im `OnError`-Event und nicht im Login-Dialog. Die neü Logik vertraut der Standard-PKI (Public CA) als Indiz für ein legitimes Renewal und pinnt das neü Cert silent neu.

Warum die Lockerung keine Sicherheitsregression ist: Der eigentliche Vertraünsanker ist *nicht* der TLS-Fingerprint, sondern der Ed25519-**Server-Signing-Key**, der pro Profil TOFU-gepinnt ist (§10.3) und bei *jeder* signierten Server-Nachricht in `RedeConnection.ReceiveLoop` gegenüber dem im Profil gespeicherten Schlüssel geprüft wird. Ein Angreifer mit gültigem CA-Cert (Sub-CA-Kompromittierung, Mis-Issuance) könnte die TLS-Schicht zwar passieren, aber keine signierten Protokollnachrichten fälschen - jede manipulierte Nachricht würde verworfen und die UI eine Signatur-Fehlermeldung anzeigen.

Self-Signed-Setups (eigener Server ohne CA-Cert): Die strikte First-Contact-TOFU-Regel bleibt erhalten - ein selbstsigniertes Cert validiert nicht über die Standard-PKI (`SslPolicyErrors != None`), fällt also nicht in den Re-Pin-Pfad. Ein Wechsel erfordert weiterhin das manülle Löschen der Pin-Datei.

Manüller Reset: Löschen von `~/.rede/.cert_pin` (oder nur des Eintrags für die jeweilige Server-URL im JSON) erzwingt ein neüs TOFU-Pinning beim nächsten Connect.

10.3 SERVER-SIGNATURVERIFIKATION

Jede Server-Nachricht wird mit Ed25519 signiert (Mechanismus in der Server-Doku §4.12 beschrieben). Auf der Client-Seite:

```
// CryptoService.VerifyServerSignature():
1. serverSig aus raw JSON extrahieren
2. serverSig-Feld per Regex aus dem JSON-String entfernen
   (NICHT re-serialisieren - JS-Schlüsselordnung muss erhalten bleiben)
3. Ed25519-Signatur über den bereinigten JSON-String verifizieren
```

Fehlende Signatur = Nachricht wird verworfen (nach TOFU-Pin).

Der Server-Signing-Key wird bei der ersten `AUTH_OK`/`REGISTER_OK` ins Profil gespeichert (`Profile.ServerSigningKey`). Jede spätere Nachricht muss mit demselben Key verifizieren. Wechselt der Server-Signing-Key (Föderations-Setup ohne `_meta`-Sync, Server-Reinstallation), bricht jede Client-Verbindung - der User muss das Profil neu anlegen oder den Eintrag manüll löschen.

`PREKEY_BUNDLE` / `USER_LOOKUP_OK`-Geräteschlüssel werden zusätzlich auf Signaturen über ihre `signedPreKey` validiert, bevor sie als Kontaktschlüssel akzeptiert werden - so wird verhindert, dass ein kompromittierter Server unerkannt einen MITM-Schlüssel als "neüs Gerät" einschmuggelt. Änderungen ausserhalb des erwarteten Signaturpfades lösen eine `[SECURITY]`-Warnung aus.

10.4 AUTOMATISCHE WIEDERVERBINDUNG

- Exponentieller Backoff (2 s Basis, 5 s für I2P)
- `volatile` Reconnect-Guard gegen gleichzeitige Reconnect-Tasks
- Events: `OnReconnecting`, `OnConnected`, `OnDisconnected`
- `ShouldReconnect`-Flag zum Deaktivieren (z.B. bei Abmeldung)

11. Sicherheitsmodell (Client)

11.1 THREAT MODEL

Bedrohung	**Schutz**
Server kompromittiert	E2EE - Server sieht nie Klartext. Sealed Sender verbirgt Absender.
Netzwerk abgehört	E2EE + TLS. I2P/Tor für IP-Verschleierung.
Vergangene Nachrichten	Forward Secrecy (Double Ratchet) - alte Schlüssel nutzlos.
Schlüssel kompromittiert	Post-Compromise Security - neür DH-Austausch heilt.
Quantencomputer (Harvest-Now-Decrypt-Later)	PQXDH - ML-KEM-768 als zweite Säule neben X25519.
Nachrichtenlänge	Festgrössen-Padding (256/1024/4096/16384).
Replay-Angriffe	NonceTracker (persistiert), Sender-Key-Zähler, server-seitige Nonces.
Swap-Auslagerung	SecureMemory.Lock() via mlock/VirtualLock.
Schlüssel im RAM	ZeroMemory nach Gebrauch, SecureTextBox für Passphrase.
Böse WebView (Group Calls)	ExternalE2EEKeyProvider isoliert den Schlüssel, kein Klartextfallback.

11.2 SICHTBARKEIT FÜR DEN SERVER (WIEDERHOLUNG AUS CLIENT-PERSPEKTIVE)

```
                                     Server sieht    Server sieht NICHT
                                     ============    ==================
Nachrichteninhalt                                    X (E2EE)
Absender (1:1)                                       X (Sealed Sender)
Empfänger (1:1)                     X (Routing)
Nachrichtengröße                                     X (Padding)
Gruppen-Mitgliedschaft              X (Roster)
Kanalnamen / Place-Profil                            X (E2EE-Metadaten)
Audio-Inhalt                                         X (SRTP / SFrame)
Anruf-Teilnehmer                    X (Signaling)
IP-Adresse (Direct)                 X
IP-Adresse (I2P/Tor)                                 X
Sozialer Graph (Kontakte)                            X (Status-Broadcast an alle)
```

Status-Updates werden vom Server an ALLE Online-Benutzer gesendet, nicht nur an Kontakte. Der Client filtert lokal. Dadurch lernt der Server nie den sozialen Graphen.

11.3 ANGEWANDTE SICHERHEITSHÄRTUNGEN

Über 12 Sicherheitsaudit-Runden. Wichtigste Massnahmen:

Kryptografie:

- Low-Order-Point-Validierung für alle DH-Operationen (7 bekannte Curve25519-Punkte)
- Ratchet-State-Rollback bei fehlgeschlagener Entschlüsselung
- Zähler-Overflow-Schutz (MaxMessageNumber = 1 Milliarde)
- Kontextgebundene Sender-Key-Signaturen (verhindert Cross-Group-Replay)
- Domain-Separation: `AUTH_CHALLENGE:`, `SEALED_SENDER_V1:`, `GROUPKEY:`, `REDE_GCALL_SFRAME_V1:`, `RedeX3DH`, `RedePQXDH`
- Deferred OTP-Verbrauch (Server poppt OTPs erst bei Nachrichtenzustellung, nicht bei Bundle-Abruf)
- HKDF-Intermediate-Zeroisierung
- Fingerprints über Base64-Darstellung (JS-v1-Kompatibilität)
- Archivierte SPKs / PQ-SPKs (5 Generationen) für Decrypt nach Rotation
- Deterministische Konvergenz crossed-X3DH-Initials per UserId-Tiebreaker

Input-Validierung:

- `ColorHelper.SafeParse()` für alle benutzerkontrollierten Farbeingaben
- DH-Header-Feldvalidierung (dh, n, pn)
- Avatar/Icon-Grössenvalidierung nach Base64-Dekodierung (`MaxAvatarBase64OnWire = 12 KiB`)
- Nonce: 24 Bytes Base64, Ciphertext: min 16 Bytes Base64
- Proxy-URL-Validierung
- Systemnachrichten-Trunkierung
- Bidi-Override-Filterung in Anzeigestrings

Speicher:

- Alle Schlüssel als `byte[]`, nicht `string` (nullbar)
- `try-finally` für alle ephemeren Schlüssel
- `Profile.ZeroSecrets()` bei Logout/Schliessung
- `SecureMemory.Lock()` für langlebige Geheimnisse
- `SecureTextBox` mit interner mlock'd Buffer
- `FlushAsync()` vor Exit

Netzwerk:

- Server-Signaturen erzwungen (keine fehlenden Sigs akzeptiert)
- TOFU Cert Pinning (mit silent Re-Pin bei CA-Renewal, siehe §10.2)
- Reconnect-Guard (volatile Flag)
- Bounded Pending Qüüs
- Generischer `[Server] …`-Banner für unbehandelte ERROR-Nachrichten (v2.19.10)

12. Build, Release und Auto-Update

12.1 BUILD-PROZESS

```
    # Linux Self-Contained
    /home/amke/.dotnet/dotnet publish
src/Rede.Desktop/Rede.Desktop.csproj \
        -c Release -r linux-x64 --self-contained \
        -p:PublishSingleFile=trü \
        -p:IncludeNativeLibrariesForSelfExtract=trü \
        -o publish/linux-x64

    # Windows Self-Contained
    /home/amke/.dotnet/dotnet publish
src/Rede.Desktop/Rede.Desktop.csproj \
        -c Release -r win-x64 --self-contained \
        -p:PublishSingleFile=trü \
        -p:IncludeNativeLibrariesForSelfExtract=trü \
        -o publish/win-x64
```

Ergebnis: Einzelnes Executable (~80-120 MB), keine .NET-Runtime erforderlich.

Build-Skripte:

- `scripts/build-release.sh` (Linux/macOS)
- `scripts/build-release.ps1` (Windows)
- `scripts/build-rnnoise.sh` (RNNoise aus Qüllen kompilieren)
- `scripts/sign-release.sh` (Release-Binärdateien signieren)

Installations-Skripte:

- `scripts/install.sh` (Linux): Klont Repo nach `~/.local/share/rede`, baut mit dotnet, erstellt Launcher in `~/.local/bin/rede` und `.desktop`-Datei mit Icon
- `scripts/install.ps1` (Windows): Äquivalentes PowerShell-Skript

12.2 RELEASE-PROZESS

1. `CurrentVersion` in `src/Rede.Core/Services/UpdateService.cs` erhöhen.
2. Für linux-x64 und win-x64 baün (§12.1).

3. Binärdateien als `REDE` / `REDE.exe` kopieren.

4. **SHA256SUMS generieren** (PFLICHT - Client verweigert Update ohne, RNNoise-Installer hard-fail ohne passenden Eintrag):(cd publish/linux-x64 && sha256sum REDE) > publish/SHA256SUMS
(cd publish/win-x64 && sha256sum REDE.exe) >> publish/SHA256SUMS
(cd src/Rede.Core/runtimes/linux-x64/native && sha256sum librnnoise.so) >> publish/SHA256SUMS
(cd src/Rede.Core/runtimes/win-x64/native && sha256sum rnnoise.dll) >> publish/SHA256SUMS

5. **Binärdateien + Native-Libs signieren** (PFLICHT - Client verweigert Update ohne `.sig`, RNNoise-Installer verweigert Schreiben ohne passende `.sig`):scripts/sign-release.sh \
  publish/linux-x64/REDE \
  publish/win-x64/REDE.exe \
  src/Rede.Core/runtimes/linux-x64/native/librnnoise.so \
  src/Rede.Core/runtimes/win-x64/native/rnnoise.dll

6. GitHub Release erstellen:gh release create v<version> \
  publish/linux-x64/REDE publish/linux-x64/REDE.sig \
  publish/win-x64/REDE.exe publish/win-x64/REDE.exe.sig \
  publish/SHA256SUMS \
  src/Rede.Core/runtimes/win-x64/native/rnnoise.dll \
  src/Rede.Core/runtimes/win-x64/native/rnnoise.dll.sig \
  src/Rede.Core/runtimes/linux-x64/native/librnnoise.so \
  src/Rede.Core/runtimes/linux-x64/native/librnnoise.so.sig \
  --target main --prerelease

7. Commit mit **NO co-authoring** und **NO AI/Claude references** in der Commit-Message.

RNNoise-Native-Libs sind optional und werden NICHT im Binary gebundelt. Die App lädt sie auf Anforderung via Settings > Install RNNoise herunter. Weil sie als nativer Code in den REDE-Prozess geladen werden, verifiziert der Installer Ed25519 `.sig` + `SHA256SUMS` vor dem Schreiben nach `~/.rede/libs/` - exakt derselbe Pfad wie der Binary-Update.

12.3 AUTO-UPDATE

```
Datei: src/Rede.Core/Services/UpdateService.cs (565 Zeilen)
```

Dualer Mechanismus:

1. **Git-basiert** (Entwickler-Installationen): `git fetch && git log` auf dem Repo
2. **GitHub Releases API** (Standalone-Exe): Prüft auf neüre Tags, lädt Binärdatei herunter

Verifikation:

- SHA256-Prüfsumme gegen `SHA256SUMS`-Asset
- Ed25519-Signatur-Verifikation gegen `.sig`-Asset
- Letztes installiertes Release-Tag wird in `~/.rede/.last-update` persistiert (verhindert Re-Prompt-Schleife)

Privilegien-Elevation (Linux):

- `NeedsRootPrivileges()` prüft Schreibzugriff auf den Binärpfad
- Falls Binary in root-eigenem Verzeichnis liegt (z.B. `/usr/bin/REDE`): Download + Verifikation als normaler User, dann `pkexec sh -c "mv ..."` für den Datei-Swap
- Falls Binary im User-Verzeichnis: direkter `File.Move`-Swap ohne Elevation

13. Abhängigkeiten

CLIENT V2 (C# / .NET 8)

Paket	Version	Verwendung
Sodium.Core	1.4.0	libsodium NaCl-Bindings
Concentus	2.2.2	Managed Opus-Codec
PortAudioSharp2	1.0.0	Cross-Platform Audio
BouncyCastle.Cryptography	2.6.2	ML-KEM-768 (PQXDH)
Avalonia	11.3.12	UI-Framework
CommunityToolkit.Mvvm	8.4.1	MVVM-Infrastruktur
Microsoft.Extensions.DI	10.0.5	Dependency Injection
WebView.Avalonia	11.0.0.1	Gruppenanrufe WebView

NATIVE LIBRARIES

Library	Lizenz	Verwendung
librnnoise.so / rnnoise.dll	BSD-3 (xiph.org)	Neuronale Rauschunterdrückung
PortAudio	MIT	Audio-I/O
libsodium	ISC	Kryptografie-Backend

CLIENT V1 (NODE.JS)

Paket	Version	Verwendung
tweetnacl	1.0.3	NaCl-Kryptografie
tweetnacl-util	0.15.1	Base64/UTF8-Utilities
ws	8.19.0	WebSocket-Client
socks-proxy-agent	8.0.5	I2P/Tor-Proxy

Stand: 2026-05-26.