Secure Overlay Chat Protocol (SOCP)

Version: 1.3 (Protocol Freeze: Week 8, Updated 17-09-2025)

Scope: Class-wide standard. All implementations MUST conform.

Topology: n-to-n mesh of Servers; Users attach to **exactly one** Local Server.

Routing model: Each Server knows its **Local Users** and a directory that maps all Users within the Network to their respective Local Server. Payloads are end-to-end encrypted and hop through Servers until they reach the User's Local Server.

Contents

1. Normative Language

- The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in RFC 2119.
- All Protocol Messages MUST be **one JSON object per line** (UTF-8, `\n` terminated).
- All Payloads carrying User content MUST be **end-to-end encrypted (E2EE) and signed**.

2. Versioning & Governance

- This document is the **single source of truth**.
- Changes require a cohort vote; if approved, bump minor version and update **freeze date**.
- All deviations from this spec are **non-compliant** and may be rejected during interop.

3. Definitions

base64url refers to the URL-safe Base 64 encoding as defined in RFC 4648

Chat refers to a place where messages can be sent either directly to a User or to multiple Users.

Client refers to a User that connects to a single Server.

Direct Message (DM) refers to a Chat between exactly two Users.

Introducer refers to a Server in the Network that introduces a new Server into the Network.

Local Server refers to the Server that a User connects to.

Local User refers to the Users that a Server directly manages.

Network refers to the decentralised secure chat network.

Payload refers to any data within a Protocol Message that is sent within the Network.

Protocol Message refers to messages between Servers that are defined in this document.

Public Channel refers to the Chat that is accessible to all Users on the Network.

Remote Server refers to a Server that the current User is not directly connected to.

Remote User refers to a User that is not directly connected to the same Server as the current User.

Server refers to each Team’s server that connects to each other within the Network.

Team refers to each group in the cohort that is implementing the protocol.

User refers to a user within the Network.

4. Cryptography

The following MUST be used:

- **Asymmetric: RSA-4096 only**
  - **Encryption:** All payloads MUST be encrypted directly with RSA-OAEP (SHA-256).
  - **Signatures:** All payloads MUST be signed using RSASSA-PSS (SHA-256).
- **Hash: SHA-256**.

- **Encodings:** Binary values (keys, ciphertexts, signatures) MUST be **base64url** (no padding) in JSON.

5. Identities & Tables

5.1. Identifiers

All unique identifiers MUST use UUID v4.

- **Server IDs:** `server_uuid`, `server_uuid`, ...
    - The Server MUST generate a UUID before joining the network.
- **User IDs:** case-sensitive strings; MUST be unique Network-wide. MUST use UUIDs.

5.2. Required In-Memory Tables (Per Server)

You are required to implement the following tables in your language of choice. The following is an implementation of the in-memory tables in Python.

```Python
servers: Dict[int, Link]                # server_id -> Link (WebSocket stream wrapper)
server_addrs: Dict[int, (host, port)]   # server_id -> advertised address
local_users: Dict[str, Link]            # user_id -> Link (to that client)
user_locations: Dict[str, str]          # user_id -> "local" | f"server_{id}"
```

6. Transport

- **WebSocket (RFC 6455)** is REQUIRED.
- Each JSON frame is sent as a **WebSocket text message** (UTF-8).
- No custom newline framing. Servers MUST parse one JSON object per WS message.
- A server MUST listen on a WS port and accept both **Server** and **User** connections.
- A connecting Server/User MUST send an **identifying first message** (see Section 8, Section 9).
- Close: use normal WebSocket closure (code 1000). Optionally send `{"type": "CTRL_CLOSE"}` before closing.
- Heartbeats MAY rely on native WS ping/pong. The `HEARTBEAT` frame (Section 8.4) remains valid for application-level checks.

7. JSON Envelope

Every Protocol Message MUST have:

```JSON
{
  "type": "STRING",       // Payload type, case-sensitive
  "from": "UUID",         // "server_id" or "user_id"
  "to":   "UUID",         // "server_id", "user_id", or "*"
  "ts":   "INT",          // Unix timetamp in milliseconds
  "payload": { },         // JSON object, payload-specific
  "sig": "BASE64URL"      // Signature over canonical payload (see §12)
}
```

- `sig` is REQUIRED on all **Server** payloads and all **User content** payloads.
- For **HELLO/BOOTSTRAP** you MAY omit `sig` if not yet possible to sign; see each type.

8. Server ↔ Server Protocol

8.1. Bootstrap (Introducer Flow)

When a new Server joins the Network, it MUST:

1. Announce itself to a trusted **Introducer**.
2. Receive a permanent `server_id` and a list of other Servers on the Network
3. Establish persistent, authenticated connections to each Server.

Static Bootstrap List

```yaml
bootstrap_servers:
  - host: "192.0.1.2"        # introducer 1
    port: 12345
    pubkey: "BASE64URL(RSA-4096-PUB)"  # pinned public key of this server

  - host: "198.50.100.3"     # introducer 2
    port: 5432
    pubkey: "BASE64URL(RSA-4096-PUB)"

  - host: "203.0.113.21"     # introducer 3
    port: 1212
    pubkey: "BASE64URL(RSA-4096-PUB)"
```

New Server → Introducer (Network Join Request):

The new Server selects an Introducer from a pre-configured bootstrap list of IPs and ports.

```json
{
  "type":"SERVER_HELLO_JOIN",
  "from":"server_id",    // your server ID
  "to":"A.B.C.D:12345", // select an introducter from static list
  "ts":1700000000000,
  "payload":{"host":"A.B.C.D","port":12345,"pubkey":"BASE64URL(RSA-4096-PUB)"},
  "sig":"..."
}
```

Notes:

- The `to` field MUST point to a known (and trusted) Introducer IP/port
- If the first Introducer is unreachable, the Server tries the next entry in its static bootstrap list.
- The static bootstrap list MUST have at least 3 servers for redundancy.

Introducer → New Server (Assignment & Server List):

Once the Introducer receives a valid join request, they will send back the following response:

```json
{
  "type":"SERVER_WELCOME",
  "from":"server_id",
  "to":"server_id",
  "ts":1700000000500,
  "payload":{
    "assigned_id": "server_id", // server_id is checked within network to verify its
    uniqueness. If it is, return same ID, otherwise return new unique ID
    "clients":[{"user_id":"user_id","host":"H1","port":P1,"pubkey":"..."},
             {"user_id": "user_id","host":"H2","port":P2,"pubkey":"..."}]
  },
  "sig":"..."
}
```

New Server → All Servers (Network Announcement):

The new Server now broadcasts its presence to all other Servers on the Network:

```JSON
{
  "type":"SERVER_ANNOUNCE",
  "from":"server_id",
  "to":"*", // Broadcast to all servers on the network
  "ts":1700000000500,
  "payload":{
    "host" "A.B.C.D", // The Server's IP
    "port" 12345,     // The Server's WS port
    "pubkey": "BASE64URL(RSA-4096-PUB)"
  },
  "sig":"..."
}
```

Servers MUST register the new Server and store `server_addrs[id]`, and verify signatures for all subsequent frames.

8.2. Presence Gossip

Advertise Local User:

When a User connects to a Server, that Server announces the User's presence to the entire Network.

The Payload MUST contain:

- The User's ID
- The ID of the Server
- The User's metadata as specified in Section 15.1.

```JSON
{
  "type":"USER_ADVERTISE",
  "from":"server_id",
  "to":"*", // Broadcast to all servers, which relays to all clients
  "ts":1700000100000,
  "payload":{"user_id":"the_user_id", "server_id":"server_id", "meta":{}},
  "sig":"..."
}
```

Processing rules:
1. Verify `sig` using from server's public key.
2. On success, update local mapping: `user_locations["user_id"] = "server_id"`
3. Forward the message to other servers (gossip).

Remove on disconnect:

When a User disconnects, the Server that they are on announces removal:

```json
{
  "type":"USER_REMOVE",
  "from":"server_1",
  "to":"*", // Broadcast to all servers, which relays to all clients
  "ts":1700000200000,
  "payload":{"user_id":"user_id","server_id":"server_id"},
  "sig":"..."
}
```

Processing rules:
1. Verify `sig`.
2. Only remove the User if the local mapping still points to that Server:

```json
if user_locations.get("user_id") == "server_id":
    del user_locations["user_id"]
```

3. Forward the removal to other Servers.

8.3. Forwarded Delivery

Deliver to a Remote User:

```json
{
  "type":"SERVER_DELIVER",
  "from":"sender_server_id",
  "to":"recipient_server_id",
  "ts":1700000300000,
  "payload":{
    "user_id":"recipient_user_id",
    "ciphertext":"<b64url RSA-OAEP(SHA-256)>",
    "sender":"Bob",
    "sender_pub":"<b64url RSA-4096 pub>",
    "content_sig":"<b64url RSASSA-PSS(SHA-256)>"
  },
  "sig":"<server_2 signature over payload>"
}
```

Routing rule:

- If `user_locations[user_id] == "local"` → deliver to local user link.
- Otherwise, it equals `server_id` → forward unchanged to `server_id`.
- Otherwise, drop and MAY emit an error upstream.

8.4. Health

Heartbeat (optional but RECOMMENDED, 15s):

You MAY implement this Protocol Message for diagnostic purposes.

```json
{
  "type":"HEARTBEAT",
  "from":"server_1",
  "to":"server_2",
  "ts":1700000002000,
  "payload":{},
  "sig":"..."
}
```

If no response has been received from a Server for **45s**, a Server SHOULD treat the connection as dead, close it, and attempt to reconnect.

9. User ↔ Server Protocol

9.1. User Hello

User → Server:

The User announces its presence to its Local Server.

```json
{
  "type":"USER_HELLO",
  "from":"user_id",  // User's ID
  "to":"server_id",  // Local Server ID
  "ts":1700000003000,
  "payload":{
    "client":"cli-v1",
    "pubkey":"<b64url RSA-4096 pub>",    // for signature verification by clients
    "enc_pubkey":"<b64url RSA-4096 pub>" // if using separate keys; else duplicate pubkey
  },
  "sig":""  // optional on first frame
}
```

Rules:

- The Server MUST reject duplicate `user_id` locally (`ERROR: NAME_IN_USE`).
- On accept: `local_users[Alice]=link; user_locations[Alice]="local"`; emit `USER_ADVERTISE` to servers.

9.2. Direct Message (E2EE)

A Direct Message is a Chat between two Users on the Network.

User → Local Server (server MUST NOT decrypt):

```json
{
  "type":"MSG_DIRECT",
  "from":"sender_user_id", // UUID of sender
  "to":"recipent_user_id", // UUID of recipient
  "ts":1700000400000,
  "payload":{
    "ciphertext":"<b64url RSA-OAEP(SHA-256) ciphertext over plaintext>",
    "sender_pub":"<b64url RSA-4096 pub of sender>",
    "content_sig":"<b64url RSASSA-PSS(SHA-256) over ciphertext|from|to|ts>"
  },
  "sig":"<optional client->server link sig; not required if TLS/Noise used>"
}
```

Server behavior:

- If `user_locations[sender_user_id] == "local"` → send `USER_DELIVER` (below) directly to the recipient.
- Otherwise, wrap as `SERVER_DELIVER` (Section 8.3) to the destination server.

Server → User (final delivery):

```
JSON
{
  "type":"USER_DELIVER",
  "from":"server_1",
  "to":"recipient_user_id",
  "ts":1700000400100,
  "payload":{
    "ciphertext":"<b64url RSA-OAEP(SHA-256)>",
    "sender":"Bob",
    "sender_pub":"<b64url RSA-4096 pub>",
    "content_sig":"<b64url RSASSA-PSS(SHA-256)>"
  },
  "sig":"<server_1 signature over payload>"  // transport integrity
}
```

Client verifies:

1. Decrypt `ciphertext` directly with the recipient's RSA-4096 private key → gets plaintext.
2. Verify `content_sig` over `(ciphertext || from || to || ts)` using `sender_pub` (RSASSA-PSS with SHA-256).

9.3. Public Channel Messaging

For simplicity in implementation, Users are added to the public channel by default and cannot be removed. Hiding of the public channel may be implemented on the Client.

See Section 15.1 for further information on the data models.

Public Channel Join

A User will join the public channel when they join the Network. The Local Server must broadcast the following messages:

```
JSON
{
  "type":"PUBLIC_CHANNEL_ADD",
  "from":"server_id",
  "to":"*",  // Broadcast to all Servers
  "ts":0,
  "payload":{"add":["Dave"],"if_version":1},
  "sig":"..."
}
```

```
JSON
{
  "type":"PUBLIC_CHANNEL_UPDATED",
  "from":"server_id",
  "to":"*",  // Broadcast to all servers
  "ts":0,
  "payload":{
    "version":2,  // Bumped every time a user is added or some other change occurs
    "wraps":[{"member_id":"id","wrapped_key":"..."},
             {"member_id":"id","wrapped_key":"..."},
             {"member_id":"id","wrapped_key":"..."},
             {"member_id":"id","wrapped_key":"..."}
    ]
  },
  "sig":"..."
}
```

Public Channel Key Distribution (Creator → Members via Servers):

```JSON
{
  "type":"PUBLIC_CHANNEL_KEY_SHARE",
  "from":"sender_server_id",
  "to":"*", // Broadcast to all servers
  "ts":1700000500000,
  "payload":{
    "shares":[
      {"member":"user_id","wrapped_public_channel_key":"<b64url RSA-OAEP(SHA-256) under
      user_id.pub>"},
      {"member":"user_id","wrapped_public_channel_key":"<b64url ...>"}
    ],
    "creator_pub":"<b64url RSA-4096 pub>",
    "content_sig":"<b64url RSASSA-PSS over SHA-256(shares|creator_pub)>"
  },
  "sig":""
}
```

Servers route each share to the correct hosting server, then to the member.

Public Channel Chat (Sender → All Members):

```JSON
{
  "type":"MSG_PUBLIC_CHANNEL",
  "from":"user_id",
  "to":"g123",
  "ts":1700000600000,
  "payload":{
    "ciphertext":"<b64url RSA-OAEP(SHA-256) ciphertext>",
    "sender_pub":"<b64url RSA-4096 pub>",
    "content_sig":"<b64url RSASSA-PSS(SHA-256) over ciphertext|from|ts>"
  },
  "sig":""
}
```

Servers fan-out to all known members' hosting servers. Servers MUST NOT decrypt.

9.4. File Transfer (DM or Public Channel)

Manifest (Sender → Server):

```JSON
{
  "type":"FILE_START",
  "from":"user_id",
  "to":"user_id",
  "ts":1700000700000,
  "payload":{
    "file_id":"uuid",
    "name":"report.pdf",
    "size":1234567,
    "sha256":"<hex>",
    "mode":"dm|public"
  },
  "sig":"<optional>"
}
```

Chunk (encrypted with same scheme as message):

```json
{
  "type":"FILE_CHUNK",
  "from":"user_id",
  "to":"user_id",
  "ts":1700000700500,
  "payload":{
    "file_id":"uuid",
    "index": 0,
    "ciphertext":"<b64url>",
  },
  "sig":""
}
```

Finish:

```json
{
  "type":"FILE_END",
  "from":"user_id",
  "to":"user_id",
  "ts":1700000701000,
  "payload":{"file_id":"uuid"},
  "sig":""
}
```

9.5. Acknowledgements & Errors

ACK (transport-level optional):

```json
{
  "type":"ACK",
  "from":"server_id",
  "to":"server_id",
  "ts":1700000800000,
  "payload":{"msg_ref":"<some id>"},
  "sig":"..."
}
```

ERROR (standardised):

```json
{
  "type":"ERROR",
  "from":"server_id",
  "to":"server_id",
  "ts":1700000900000,
  "payload":{"code":"USER_NOT_FOUND","detail":"Bob not registered"},
  "sig":"..."
}
```

Error codes (implement parsing): `USER_NOT_FOUND`, `INVALID_SIG`, `BAD_KEY`, `TIMEOUT`, `UNKNOWN_TYPE`, `NAME_IN_USE`.

10. Routing Algorithm (Authoritative)

Given `route_to_user(target_u, frame)`:

1. If `target_u in local_users` → send directly (`USER_DELIVER`).
2. Otherwise, if `user_locations[target_u] == "server_id"` → send (`SERVER_DELIVER`) to `servers[id]`.
3. Otherwise, emit `ERROR(USER_NOT_FOUND)` to the **originating** endpoint.

Servers MUST NOT loop messages. Each Server MUST keep a short-term `seen_ids` cache for server-delivered frames (by `(ts,from,to,hash(payload))`) and drop duplicates.

11. Heartbeats & Timeouts

- Send `HEARTBEAT` every **15s** to all Servers.
- If **45s** without any frame from a Server ⟶ mark connection as dead, close, and try reconnecting (using `server_addrs`).
- On connection loss, User presence may become stale. Implementations SHOULD lazily correct presence when deliveries fail or when new gossip is received.

12. Signing & Verification (Canonical)

The double pipes (`||`) in this case means "OR".

- **Content signature** (`content_sig`) covers **only end-to-end fields**:
  - For DM: `SHA256(ciphertext || from || to || ts)`
  - For Public Channel: `SHA256(ciphertext || from || ts)`
  - For Public Channel Key Share: `SHA256(shares || creator_pub)`
- **Transport signature** (`sig` in envelope) covers **`payload` object only** (canonicalised with JSON key sort; no whitespace variation).
- **Key sources**:
  - User pubkeys fetched from **Server Database** (or supplied in `USER_HELLO`, subject to directory verification).
  - Server pubkeys exchanged at bootstrap and pinned to `server_id`.

13. Server Database (Login & Keys)

- **Directory functions** (out of band to Network protocol, but REQUIRED cohort-wide):

    - Register `user_id` with **RSA-4096 public key**.
    - Serve authenticated queries: `get_pubkey(user_id)` returns pubkey + signature by directory.
    - Optionally store **revocation** and **rotation** metadata.
- **User login** model (recommended):
    - Client encrypts its RSA private key locally with a password-derived key.
    - On login, client decrypts locally; proves possession by **signing a nonce** from directory.

14. Mandatory Features (Interoperability)

Implementations MUST support the following Client commands:

- `/list` → server returns **sorted** list of known online users.
- `/tell <user> <text>` → DM using RSA-4096
- `/all <text>` → Broadcast a message to the public channel.
- `/file <user> <path>` → file transfer (manifest + encrypted chunks).

Servers MUST:

- Accept bootstrap & link other servers.
- Gossip `USER_ADVERTISE` / `USER_REMOVE`.
- Route `SERVER_DELIVER` without decrypting payloads.

15. Server Database: Users, Profiles, Public Channel

Each Team's server MUST have its own **persistent** database.

The exact model of the database is up to each Team's implementation. However, the following fields MUST be implemented at a minimum.

15.1. Data Model

Users

```sql
users(
  user_id TEXT PRIMARY KEY, -- UUID, use UUID type if supported (e.g., in PostgreSQL)
  pubkey TEXT NOT NULL, -- RSA-4096 (base64url)
  privkey_store TEXT NOT NULL, -- Encrypted private key blob
  pake_password TEXT NOT NULL, -- PAKE verifier / salted hash
  meta TEXT, -- Optional decorative fields, use JSONB type if supported
  version INT NOT NULL -- bumps on deco/security changes
)
```

Below are the optional keys for `meta`:

```json
{
  "display_name": "Alice",
  "pronouns": "she/her",
  "age": 37,
  "avatar_url": "",
  "extras": {}
}
```

The routing and cryptography MUST NOT depend on `meta`.

Public Channel

```sql
-- You can support groups, but only the public channel is REQUIRED
groups(
  group_id TEXT PRIMARY KEY, -- Public channel should be called "public", UUIDs otherwise
  creator_id TEXT NOT NULL, -- Public channel creator_id is "system"
  created_at INT, -- Use dedicated timestamp times if your flavour of SQL supports it
  meta TEXT, -- Optional: title, avatar, extras; Use JSONB type if supported
  version INT NOT NULL -- Bumps on membership/key rotation
)

group_members(
  group_id TEXT NOT NULL, -- UUID or "public" for public channel
  member_id TEXT NOT NULL, -- UUID
  role TEXT, -- "owner" | "admin" | "member", use enum type if supported. Public channel only
  has "member"
  wrapped_key TEXT NOT NULL -- RSA-OAEP(SHA-256) of current group_key for member_id
  added_at INT, -- Timestamp of when user was added
  PRIMARY KEY (group_id, member_id)
)
```

The group key is a random 256-bit value per `groups.version`. The database stores only per-member wraps; never the clear group key in replies.

Notes for public broadcast channel:

- The `group_id` for the public channel is `public`
- The `creator_id` is `system`
- `created_at` should be when the server joins the network and the public channel is made known to them
- Every member of the public channel is only a `member` (i.e., there are no owners or admins)

15.2. Label Fallbacks (Display Only)

For ease of use, you SHOULD implement label fallbacks with the following hierarchy:

1. `meta.display_name`
2. Otherwise `user_id` or `group_id`

16. Backdoors (Assignment Requirement)

Each Team MUST intentionally include **at least 2 vulnerabilities** in the **backdoored** submission:

Non-exhaustive list of allowed examples (non-destructive, in-scope):

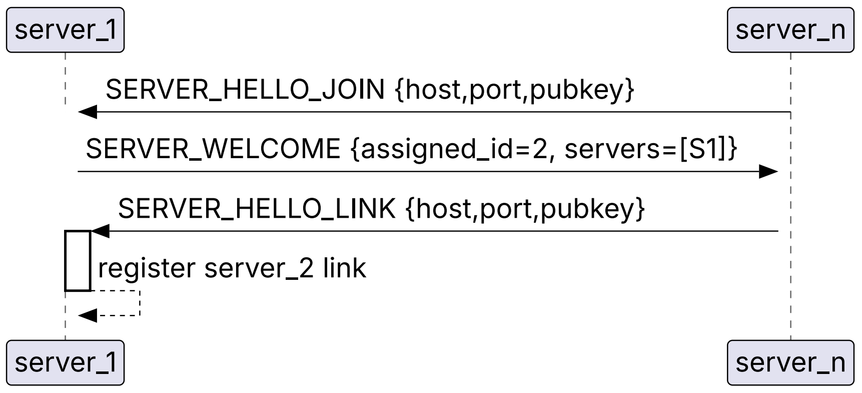
- Accepting **RSA-1024** keys (weak) while claiming 4096.
- Accepting keys with weak parameters (e.g. unusually small public exponent) or malformed but still treated as valid.
- Missing duplicate-message suppression → replay acceptance.
- Trusting unauthenticated `USER_ADVERTISE`.

Prohibited: anything that escapes VM, exfiltrates real data, or harms the host.

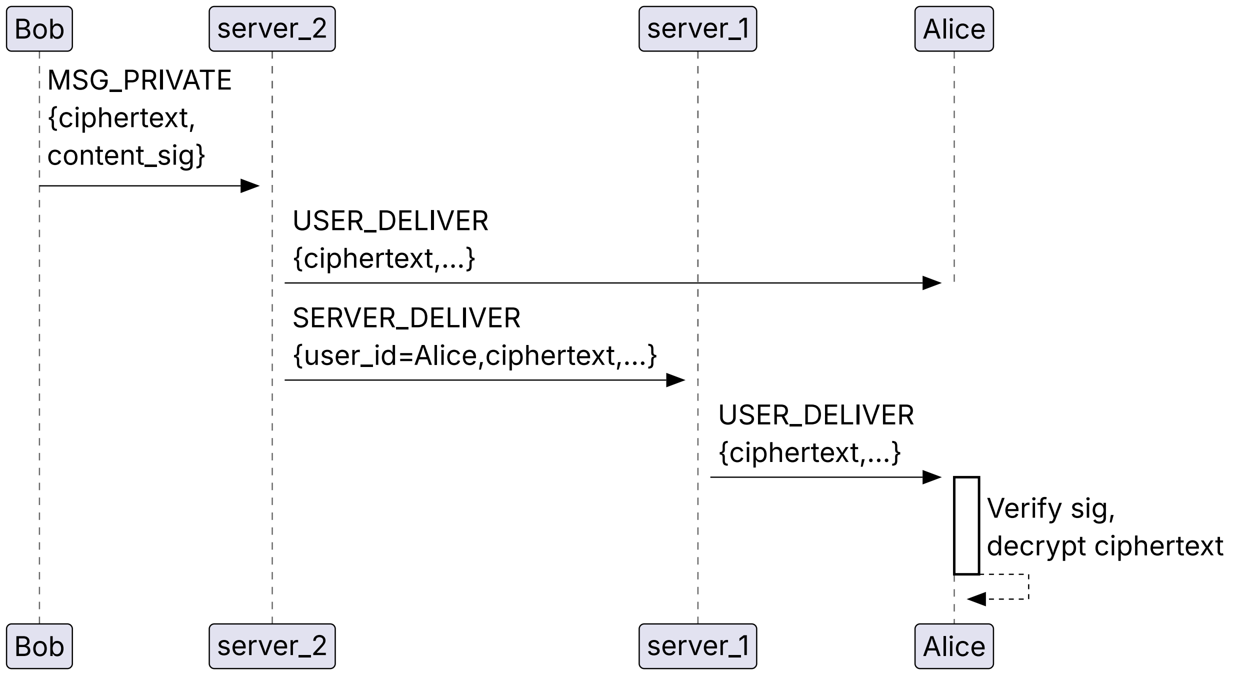
17. Sequence Diagrams

These diagrams are to aid your understanding of the protocol.

17.1. Bootstrap


server_1
server_n
SERVER_HELLO_JOIN {host,port,pubkey}
SERVER_WELCOME {assigned_id=2, servers=[S1]}
SERVER_HELLO_LINK {host,port,pubkey}
register server_2 link
server_1
server_n

17.2. Direct Message (Bob → Alice)


Bob
server_2
server_1
Alice
MSG_PRIVATE {ciphertext, content_sig}
USER_DELIVER {ciphertext,...}
SERVER_DELIVER {user_id=Alice,ciphertext,...}
USER_DELIVER {ciphertext,...}
Verify sig, decrypt ciphertext
Bob
server_2
server_1
Alice

18. Compliance Checklist

- [ ] RSA-4096 keys; RSA-OAEP (SHA-256) for encryption; RSASSA-PSS (SHA-256) for signatures.
- [ ] All user content carries `content_sig` (end-to-end).
- [ ] All server frames carry transport `sig`.
- [ ] `USER_ADVERTISE` / `USER_REMOVE` implemented.
- [ ] `SERVER_DELIVER` routing implemented with loop suppression.
- [ ] Heartbeats (15s) and 45s timeout.
- [ ] Error codes implemented.
- [ ] `/list`, `/tell`, `/all`, `/file` supported.
- [ ] README with run commands and dependencies.

19. Changelog

Version	**Changes**
v1.0	Initial proposal
v1.1	• Use WebSockets instead of TCP • Addition of master database for users, profiles, and groups
v1.2	• Added definitions section • Consolidated language and terminology • Added public channels and remove groups • Use UUIDs as unique identifiers for Users and Servers • Removed master database • Added more information on Server bootstrapping process • Added changelog section
v1.3	• Replaced all AES-256-GCM encryption with RSA-4096 only. • Removed AES IVs, tags, and wrapped AES keys from Direct Messages, Public Channels, and File Transfers. • Updated `content_sig` handling to cover only RSA-encrypted payloads. • Updated compliance checklist, mandatory features, and diagrams to reflect RSA-only design.