HandleComms

Design Document
April 2025

Jesse Anderson

Contents:

Project Overview

This project aims to provide a completely decentralized communications system focused mainly on **Cardano** DApp messaging and the **HandleChat** platform. This project defines the architecture and implementation plan for a **decentralized, end-to-end encrypted messaging system** built atop peer-to-peer (P2P) networking, the Signal Protocol, and blockchain-based cryptographic identity.
The system is designed for **privacy, resilience, and autonomy**, providing users with secure communication without relying on centralized servers or trusted intermediaries. It is architected to **resist** *surveillance, censorship, and tampering* while remaining user-friendly and developer-extensible. Its foundation in **Rust** and modern decentralized tooling makes it suitable for integration into both consumer applications and enterprise-grade secure communication platforms, leveraging its memory safety, performance, and cross-platform capabilities.
Rust is chosen for its suitability in building secure and efficient native binaries that can be easily distributed and run on a wide range of systems, including mobile devices, desktop clients, and edge nodes.

At the core of the system are three key **Rust** libraries:

- **`libp2p`** – Enables robust, modular P2P networking with NAT traversal, pub-sub messaging, and peer discovery, forming the decentralized communication backbone.
- **`libsignal`** – Provides end-to-end encrypted session management using the Double Ratchet Algorithm, ensuring forward secrecy, authentication, and message confidentiality.
- **`rustcrypto`** – Supplies audited, safe, and performant cryptographic primitives to support blockchain-compatible identity, message signing, and secure hashing.

The messaging system supports a comprehensive feature set:

- **Decentralized & Incentivized Peer Nodes** for resilient, censorship-resistant routing.
- **Blockchain Address-Based Identity**, users connect via cryptographically verifiable wallets.
- **Compromise Resistance** via Signal's "Double Ratchet" and Rust's memory safety.
- **End-to-End Encryption** that is persistent, zero-trust, and cross-platform.
- **Ephemeral Messaging** with secure, in-memory handling and optional message expiration.
- **Contact Requests, Whitelisting, and Blacklisting** to establish controlled trust boundaries.
- **Jurisdiction Resistance** using message splitting and obfuscation at the transport layer.
- **Direct Peer-to-Peer Communication** via `libp2p`, avoiding centralized relay servers.
- **Persistent Encrypted Message History**, stored locally and securely.
- **Blockchain Validated Messages** for verifiable timestamps, origin, and content hashes.
- **Certified Delivery** via blockchain signatures of sender and recipient.

Functional Requirements

These requirements describe what the system must do to fulfill user and protocol expectations. They are expressed in terms of verifiable actions or system outputs under defined conditions.

Identity & Authentication

- Generate or import a persistent identity from a blockchain address
- Authenticate contacts through signature verification using the appropriate chain and cryptographic scheme
- Prevent impersonation and unauthorized access

Messaging

- Send and receive encrypted messages between verified peers
- Establish secure sessions with perfect forward secrecy (Double Ratchet)
- Deliver messages through direct or relayed libp2p paths
- Support ephemeral (auto-expiring) and persistent message types

Contact Management

- Send, accept, and reject connection (friend/contact) requests
- Maintain a whitelist/blacklist for filtering message sources
- Prevent unwanted communication from blocked contacts

Message Splitting

- Split encrypted messages into multiple fragments
- Ensure no node can store more than one fragment
- Reassemble fragments only at the intended destination

Blockchain Integration

- Sign messages and requests with the user's blockchain private key
- Verify on-chain signatures to prove identity
- Optionally anchor message hashes or delivery proofs to blockchain for immutability

Storage & History

- Store messages locally in an encrypted inbox with optional persistence
- Allow users to purge stored messages by age, size, or metadata
- Enable local search and message metadata indexing

Non-Functional Requirements

These requirements define how the system performs and what qualities it must exhibit beyond its core functions.

Security

- Use strong, well-reviewed cryptographic primitives (rustcrypto, libsignal)
- Maintain forward secrecy, message authentication, and plausible deniability
- Prevent metadata leakage wherever feasible (e.g., through message splitting and obfuscation)

Privacy

- Default to end-to-end encryption and local message storage
- Avoid centralized servers and single points of failure
- Allow users to operate fully pseudonymously

Performance

- Low-latency message exchange over libp2p direct or relayed links
- Resilient under intermittent connectivity and mobile devices
- Efficient reassembly of message fragments

Scalability

- Support thousands of nodes using pub-sub, DHT, and gossipsub routing
- Leverage Rust’s async runtime and minimal runtime overhead for peer scaling
- Allow future integration of storage delegation or data swarming

Usability

- Simple CLI or GUI interface with minimal setup steps
- Allow users to bootstrap with a blockchain address and peer bootstrap list
- Provide meaningful errors, confirmations, and timeouts for message events

Extensibility

- Modular protocol stack with clearly defined API boundaries
- Support plugin modules for new cryptographic curves, blockchains, transports
- Allow easy integration of DAO-based governance or incentive systems in future iterations

API Design

The system will expose a clean, Rust-native API for developers, which can be wrapped via FFI for other languages or embedded in CLI/GUI clients. The API will emphasize type safety, clear error handling, and cryptographic correctness.

Identity API

```rust
fn generate_identity() -> Identity;
fn import_identity(secret: &[u8]) -> Result<Identity, Error>;
fn sign_message(message: &[u8]) -> Signature;
fn verify_signature(message: &[u8], sig: Signature, address:
BlockchainAddress) -> bool;
```

Messaging API

```rust
fn send_message(to: PeerId, plaintext: &[u8], options:
MessageOptions) -> Result<DeliveryReceipt, Error>;
fn receive_messages() -> Vec<DecryptedMessage>;
fn mark_as_read(message_id: Uuid);
fn delete_message(message_id: Uuid);
```

Session & Encryption API

```rust
fn establish_session(peer: PeerId) -> Result<(), Error>;
fn encrypt_payload(to: PeerId, plaintext: &[u8]) -> EncryptedPayload;
fn decrypt_payload(from: PeerId, encrypted: EncryptedPayload) ->
Result<Vec<u8>, Error>;
```

Contact & Social API

```rust
fn send_contact_request(to: BlockchainAddress) -> Result<(), Error>;
fn accept_contact_request(peer: PeerId) -> Result<(), Error>;
fn block_contact(peer: PeerId);
fn get_contacts() -> Vec<Contact>;
```

Fragmentation API (optional)

```
fn fragment_encrypted_message(msg: &[u8], count: u8) ->
Vec<MessageFragment>;
fn reassemble_fragments(fragments: Vec<MessageFragment>) ->
Result<Vec<u8>, Error>;
```

Blockchain Anchoring API (optional)

```
fn anchor_message_hash(hash: [u8; 32]) -> TransactionReceipt;
fn validate_anchored_hash(hash: [u8; 32], proof: BlockchainProof) ->
bool;
```

Language Considerations

After evaluating the requirements for decentralized peer-to-peer (P2P) messaging, cryptographic rigor, cross-platform support, and ease of distribution, **Rust** is the recommended programming language for implementing the core messaging protocol.

- **Security and Memory Safety**
  Rust's ownership model and strict compile-time checks eliminate entire classes of bugs such as null pointer dereferencing and data races. This is critical for a system dealing with sensitive encrypted communication and network input.

- **Cryptography Ecosystem**
  Rust has a growing ecosystem of mature, audited cryptographic libraries that support both modern blockchain cryptography (e.g., secp256k1, ed25519, BLS) and the Signal Protocol primitives (e.g., X25519, AES-GCM, HMAC). Native bindings are also available for libsodium and libsignal, facilitating integration where needed.

- **Cross-Platform Native Binaries**
  Rust compiles to efficient native code across major operating systems (Linux, macOS, Windows) and architectures (x86_64, ARM), enabling simple, dependency-free binary distributions. This is ideal for nodes that require straightforward installation and execution.

- **Concurrency and Networking**
  Built-in async/await support, paired with high-performance async runtimes (e.g., Tokio), makes Rust suitable for writing robust and scalable P2P networking stacks.

- **Growing Adoption in Decentralized Systems**
  Rust is already used in major blockchain and distributed system projects (e.g., Polkadot/Substrate, Solana, libp2p), which aligns with the decentralized goals of this system and provides an ecosystem of compatible libraries and tools.

- **Community and Maintenance**
  The language has a vibrant and security-conscious community, with frequent updates and excellent tooling (e.g., Cargo, Clippy, Rustfmt) that improve developer productivity and maintainability.

Alternative Considerations

- **Go** offers simplicity and fast development but lacks the same level of cryptographic and memory safety assurances as Rust.

- **C/C++** are performant and flexible but come with higher risks for memory-related vulnerabilities and more complex dependency management.

- **Node.js or Python** provide fast prototyping but are unsuitable for native binary distribution and do not meet the performance or security requirements of this system.

Given the critical need for secure communication, lightweight deployment, and native performance, **Rust is the language of choice** for implementing the core protocol logic of this decentralized messaging system.

Key Libraries

To support the implementation of a secure, decentralized messaging protocol, several Rust packages stand out as essential building blocks. These libraries provide mature, performant, and well-maintained abstractions for networking, encryption, and protocol logic.

`libp2p` – Peer-to-Peer Networking Stack

- **Decentralized Networking Foundation**
  `libp2p` is a modular and extensible networking stack designed specifically for peer-to-peer applications. Originally developed as part of the IPFS project, it has a strong focus on decentralization, NAT traversal, transport abstraction, and peer discovery.

- **Transport Agnostic**
  Supports multiple transport protocols (TCP, QUIC, WebSockets, etc.), allowing dynamic adaptation based on environment and network topology.

- **Built-In Protocols**
  Includes essential P2P capabilities out of the box—such as pub-sub (Gossipsub), DHT for peer routing, and relay protocols—saving significant development time.

- **Well-Aligned with Decentralized Use Cases**
  Used in large-scale blockchain and distributed projects (e.g., Polkadot/Substrate), making it a natural fit for this messaging system's architecture.

`libsignal` – Signal Protocol Implementation

- **End-to-End Encryption Standard**
  `libsignal` provides the core primitives and session management needed for the Signal Protocol, a proven standard in secure messaging used by Signal, WhatsApp, and others.

- **Forward Secrecy and Deniability**
  Supports key Signal Protocol features such as Double Ratchet, pre-keys, and message authentication, ensuring privacy even in the face of key compromise.

- **Cross-Language Compatibility**
  While a native Rust implementation is still evolving, Rust bindings to official

C implementations are available and integrate well with Rust's FFI system.

- **Foundation for Trust**
  Establishes the end-to-end encrypted layer critical to user privacy and protocol trustworthiness.

`rustcrypto` – Cryptographic Primitives

- **Modular and Audited**
  `rustcrypto` is a collection of pure-Rust cryptographic libraries maintained under a unified organization. It includes implementations of widely used algorithms such as SHA-2, AES, HMAC, and curve25519.

- **No Unsafe Code**
  Many components are written entirely in safe Rust, minimizing the attack surface and avoiding undefined behavior.

- **High Interoperability**
  Compatible with many external cryptographic systems and protocols, including those used in blockchain (e.g., ed25519, secp256k1).

- **Actively Maintained and Audited**
  The project has a strong contributor base and is undergoing continual review, making it a stable and secure choice for core cryptographic operations.

The selected Rust packages—`libp2p`, `libsignal`, and `rustcrypto`—play complementary roles in forming a robust and modular architecture for the decentralized messaging system. Their integration ensures that the system is secure, extensible, and aligned with the core goals of decentralization and privacy.

Layered Architecture Integration

1. **Networking Layer – `libp2p`**
   Acts as the foundation of the node-to-node communication stack. It manages peer discovery, message routing, NAT traversal, and transport abstraction. This ensures reliable and censorship-resistant connectivity between nodes, regardless of their environment.

2. **Encryption and Messaging Layer – `libsignal`**
   Operates on top of `libp2p`, providing end-to-end encryption between peers. `libsignal` handles session establishment, identity verification, message ratcheting, and key rotation, ensuring forward secrecy and deniability. Messages are transmitted over `libp2p` channels but are encrypted and authenticated independently using Signal's protocol logic.

3. **Cryptographic Primitives – `rustcrypto`**
   Supports both `libp2p` and `libsignal` by supplying the low-level cryptographic operations such as hashing, symmetric encryption, public key generation, and digital signatures. `rustcrypto` provides safe and efficient implementations of these primitives, reinforcing security without relying on native C code.

Benefits of Modular Integration

- **Security by Design**
  Each package adheres to modern cryptographic standards and benefits from Rust's compile-time guarantees. This minimizes runtime vulnerabilities and supports formal auditing of individual components.

- **Scalability and Maintainability**
  Decoupling the networking, encryption, and cryptography layers allows the system to evolve independently. For example, switching out a cryptographic primitive (e.g., to a post-quantum variant) or introducing a new transport protocol requires minimal changes to adjacent layers.

- **Extensibility**
  Future features such as metadata protection, anonymous routing (e.g., mixnets or onion routing), or integration with smart contract identity systems can be layered in with minimal disruption to existing code.

- **Cross-Platform Compatibility**
  All selected packages compile cleanly across desktop, server, and embedded environments, making it feasible to run full or lightweight nodes in diverse deployment scenarios—from smartphones to dedicated servers.

Jurisdiction-Resistant Message Splitting

To enhance censorship resistance and protect user privacy, the system supports optional **post-encryption message splitting**, wherein each message is fragmented into multiple encrypted parts distributed across unrelated nodes. No single node has access to more than one fragment, significantly reducing the surveillance or seizure value of any given node's data.

Overview

- **Fragmentation Process:**
  - Once a message is encrypted end-to-end, it is divided into fragments using a deterministic or secret-sharing scheme (e.g., standard chunking, Shamir's Secret Sharing, or erasure coding).
  - Each fragment is tagged with metadata to facilitate reassembly: message ID, part index, total parts, and integrity hash.

- **Storage & Routing:**
  - Fragments are routed and stored on distinct peers, selected via a DHT, distance metric, or randomized gossip.
  - Delivery involves reassembling fragments at the recipient node before decryption.

- **Reassembly Guarantee:**
  - A message is only reconstructable at the destination node, which holds the decryption key and collects all parts.
  - Optional timers or ephemeral expiration metadata prevent long-term storage.

Benefits

- **Jurisdiction Resistance:**
  - No node can reconstruct the message independently, making subpoenas or node compromise minimally useful.
  - Increases resilience against legal attacks on infrastructure in high-risk jurisdictions.

- **Obfuscation of Message Flow:**
  - Fragment distribution and indirect routing complicate traffic analysis, even when metadata-free transmission is not feasible.

- **Decoupling of Routing and Identity:**
  - Nodes storing fragments don't know the sender, receiver, or even the context of the message.

Trade-Offs

- **Increased Complexity:**
  - Fragmentation, routing, and reassembly introduce additional coordination and failure recovery logic.

    - Handling partial delivery and retry mechanisms becomes more involved.

- **Latency and Bandwidth Overhead:**
    - Fragmented messages require more round-trips and bandwidth due to multi-part delivery and redundancy needs.
    - Recipient must wait for all fragments before decryption.

- **Loss Recovery:**
    - Fragment loss requires timeouts, retransmissions, or parity redundancy (e.g., using erasure coding like Reed-Solomon).
    - Adds pressure to implement reassembly timeouts and chunk caching.

Implementation Considerations

- **Fragmentation Strategy Options:**
    - Simple Chunking: Divide encrypted ciphertext into equally sized pieces.
    - Redundant Coding: Use Reed-Solomon or fountain codes for fault tolerance.
    - Shamir Secret Sharing (if desired): Provides k-of-n recoverability, but is computationally heavier.

- **Storage Expiry & Privacy:**
    - Fragments can be time-limited with TTL metadata or stored only until delivery is confirmed.
    - Nodes should have no incentive or means to store fragments beyond usefulness.

- **Libp2p Extensions:**
    - Fragment gossip and routing can extend libp2p's pub-sub or Kademlia systems.
    - Future work may involve support for fragment-level onion routing.

Peer Node Discovery

Given the decentralized peer-to-peer (P2P) nature of the messaging system, an efficient and robust node peer discovery mechanism is crucial. This section outlines the considerations and proposed techniques for enabling nodes to find and connect with each other within the network.

Requirements

- **Decentralization:** The discovery mechanism must operate in a decentralized manner, avoiding reliance on centralized servers or services to maintain network resilience and prevent single points of failure.
- **Scalability:** The system should efficiently support a large number of nodes, allowing for network growth without significant performance degradation in the discovery process.
- **Dynamism:** The discovery process must adapt to nodes joining and leaving the network frequently, ensuring that nodes maintain an accurate view of the active peers.
- **Security:** Discovery mechanisms should be secure to prevent malicious nodes from manipulating the process, such as injecting false peer information or partitioning the network.
- **Efficiency:** Minimize the overhead associated with peer discovery to conserve network bandwidth and processing resources, especially important for mobile or resource-constrained devices.

Proposed Techniques

Considering the requirements and the system's reliance on Rust for implementation, the following peer discovery techniques are suitable:

1. **Distributed Hash Table (DHT):**
   - A DHT, such as Kademlia, provides a decentralized lookup service where nodes collectively maintain information about the location of other nodes and resources.
   - Rust libraries like can be leveraged to implement DHT functionality.
   - DHTs offer excellent scalability and dynamism, making them suitable for a growing and changing P2P network.
   - Security considerations, such as against Sybil attacks, must be addressed through proper implementation and security practices.
2. **Multicast DNS (mDNS):**
   - mDNS is suitable for local network discovery, allowing nodes to easily find peers within the same network segment.
   - Rust crates are available to implement mDNS.
   - While mDNS is not suitable for wide-area P2P networks, it can complement other discovery methods for enhanced local connectivity.
3. **Bootstrap Nodes:**
   - A set of well-known, stable nodes can serve as initial contact points for new nodes joining the network.

    - These bootstrap nodes provide the initial peer list, enabling new nodes to bootstrap into the DHT or other discovery mechanisms.
    - The number of bootstrap nodes should be sufficiently large and distributed to avoid bottlenecks and single points of failure.
4. **Peer Exchange (PEX):**
    - Nodes actively exchange known peer information with each other.
    - This gossip-style protocol helps disseminate peer information throughout the network, improving connectivity and resilience.
    - PEX can be combined with other discovery methods to maintain an up-to-date view of the network.

Hybrid Approach

A combination of these techniques is recommended to achieve optimal peer discovery:

- New nodes use bootstrap nodes to join the DHT.
- DHT provides the primary mechanism for discovering peers across the wider network.
- mDNS facilitates discovery of peers within the local network.
- PEX is used to continuously update peer information and enhance network connectivity.

Security Considerations

- **Node Authentication:** Implement strong node authentication mechanisms to prevent unauthorized nodes from joining the network or injecting false peer information.
- **Data Validation:** Validate all peer information received from other nodes to ensure its integrity and prevent malicious manipulation.
- **Attack Resistance:** Design the discovery mechanism to be resilient against common P2P attacks, such as Sybil attacks, Eclipse attacks, and Distributed Denial of Service (DDoS) attacks..

Signal Protocol Overview

The Signal Protocol is a cryptographic protocol that provides end-to-end encryption for instant messaging. It combines several cryptographic techniques to achieve its security goals:

- **Key Agreement:** The Elliptic Curve Diffie-Hellman (ECDH) key agreement protocol is used to establish shared secrets between communicating parties.
- **Encryption:** Symmetric encryption using AES-256 is employed to encrypt the message content.
- **Message Authentication:** HMAC-SHA256 is used to provide message authentication and integrity.
- **Double Ratchet Algorithm:** This algorithm manages key updates, providing forward secrecy (protection of past communications if keys are compromised) and future secrecy (protection of future communications if keys are compromised).

Handshake Protocol

The Signal Protocol requires an initial key exchange handshake before secure messaging can occur. This handshake establishes the initial shared secrets necessary for encryption. The handshake involves the following steps:

- **Identity Key Exchange:** Each user has a long-term identity key pair. These public keys must be securely exchanged or verified through an out-of-band mechanism to establish trust.
- **Pre-key Exchange:** One party generates a set of pre-key pairs and publishes the public parts. The other party retrieves one of these pre-keys to perform an ECDH key agreement.
- **Ephemeral Key Agreement:** Both parties generate ephemeral key pairs for each session and perform ECDH key agreements.
- **Shared Secret Derivation:** The ECDH outputs are combined to derive shared secrets used for message encryption.

Message Format

Signal Protocol messages can be of different types, each with a specific format:

- **PreKey Message:** This message type is used during the initial handshake. It includes the sender's identity key, ephemeral key, and the recipient's pre-key ID. It is used when the sender does not have a session established with the receiver.
- **Message:** This is the standard message type for encrypted communication after the handshake. It contains the ephemeral key and the ciphertext.
- **Private Key Message:** This message type is used to transmit encrypted messages when a session has already been established. It is more compact as it omits certain handshake data.

The message format typically includes:

- **Type Identifier:** Indicates the message type (PreKey Message, Message, or Private Key Message).
- **Key Material:** Ephemeral public keys, pre-key IDs, or other keying information.
- **Ciphertext:** The encrypted message content.
- **Message Authentication Code (MAC):** Used for message integrity and authentication.

Protocol Flow

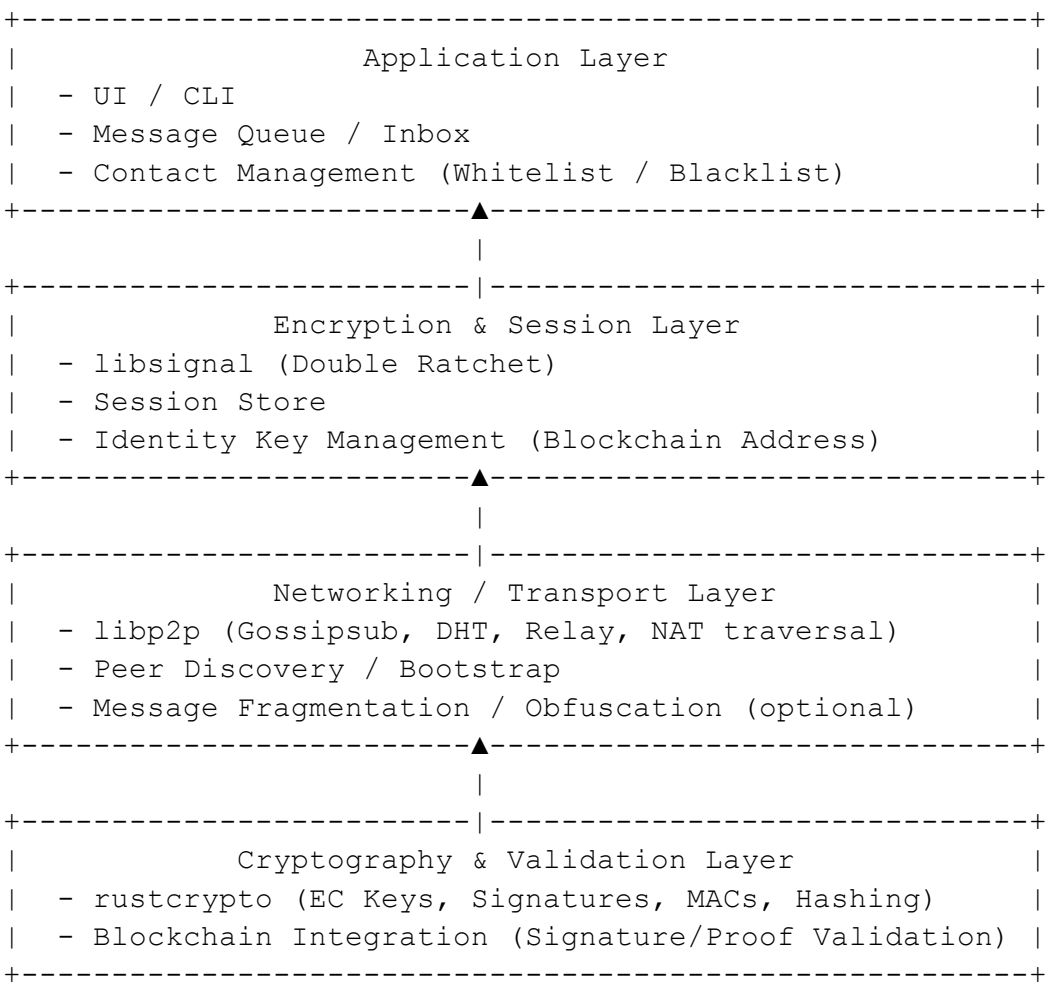
1. **Session Establishment:**
    - The recipient generates an identity key, a signed pre-key, and a bundle of one-time pre-keys.
    - The sender retrieves the recipient's pre-key bundle.
    - The sender initiates the handshake, creating a PreKey Message.
    - Both parties derive shared secrets.
    - A secure session is established.

2. **Message Transmission:**
    - Once a session is established, parties use the Double Ratchet algorithm to derive new shared keys for each message.
    - Messages are encrypted using AES-256 with the derived keys.
    - Each message includes a MAC for authentication.

3. **Key Rotation:**
    - The Double Ratchet algorithm ensures that keys are continuously rotated, providing forward and future secrecy.
    - This rotation happens with every message sent and received.

System Architecture

The decentralized messaging system is built in a modular, layered architecture that promotes security, testability, and extensibility. Each layer is responsible for a specific set of concerns, providing a clear separation of responsibilities while enabling plug-and-play component design.



Application Layer

- **Responsibilities:**
  - User interface (CLI or GUI)
  - Message queueing and inbox management
  - Contact management with whitelisting and blacklisting

- **Design Rationale:**
  - Keeps user interactions isolated from core logic
  - Easily testable through mock UIs and interaction scenarios

Encryption & Session Layer

- **Components:**
  - libsignal: Handles Double Ratchet algorithm for forward secrecy and deniability
  - Session Store: Manages ratchet states and identity keys
  - Identity Key Management: Uses blockchain addresses as persistent identities

- **Security Features:**
  - End-to-end encryption
  - Resistance to key compromise (Perfect Forward Secrecy)
  - Verified identity binding via on-chain signatures

Networking / Transport Layer

- **Components:**
  - libp2p: Provides peer discovery, message relay, and NAT traversal
  - Gossipsub / DHT: Used for routing, relay fallback, and decentralized bootstrapping
  - Message Fragmentation: Optional mechanism for message splitting to support jurisdiction resistance

- **Design Goals:**
  - Eliminate centralized servers
  - Ensure robust peer connectivity across varied networks
  - Obfuscate traffic flow patterns

Cryptography & Validation Layer

- **Components:**
  - rustcrypto: Cryptographic primitives (ECDSA, AES-GCM, ChaCha20, HMAC, etc.)
  - Blockchain Integration:
    - Address validation and signature checks
    - Message anchoring and certified delivery

- **Benefits:**
  - Hardened cryptographic foundation
  - Leverages Rust's compile-time safety and minimal unsafe code usage
  - Allows decentralized reputation, delivery verification, and identity proofing

Revisions

v1.0: Initial design

Revision date: 2025-04-10

Disclaimer

This document is intended to describe the planned software architecture of the HandleComms system as of the date of publication. It serves as a conceptual and technical guide for understanding the design choices, component interactions, and system boundaries.

Please note that while every effort has been made to ensure the accuracy and completeness of the architectural design described herein, the actual implementation of the system may deviate from this design due to evolving business requirements, technical constraints, implementation decisions, performance considerations, or other factors encountered during the software development lifecycle.

This document does not constitute a commitment to deliver the software exactly as described, and the authors make no representations or warranties, express or implied, regarding the final implementation, fitness for a particular purpose, or the absence of defects. Use of this document is at your own risk.

The information in this document is provided "as is" and may be subject to change without notice.

THIS REPORT MAY NOT BE TRANSMITTED, DISCLOSED, REFERRED TO, MODIFIED BY, OR RELIED UPON BY ANY PERSON FOR ANY PURPOSES WITHOUT THE Kora Labs' PRIOR WRITTEN CONSENT.

THIS REPORT IS NOT, NOR SHOULD BE CONSIDERED, AN ENDORSEMENT, APPROVAL OR DISAPPROVAL of any particular project, team, code, technology, asset or anything else. This report is not, nor should be considered, an indication of the economics or value of any technology, product or asset created by any team or project used in this design. THIS REPORT DOES NOT PROVIDE ANY WARRANTY OR GUARANTEE REGARDING THE QUALITY OR NATURE OF THE TECHNOLOGY ANALYZED, nor does it provide any indication of the technology's proprietors, business, business model or legal compliance.

To the fullest extent permitted by law, the Kora Labs DISCLAIMS ALL WARRANTIES, EXPRESSED OR IMPLIED, IN CONNECTION WITH THIS REPORT, ITS CONTENT, AND THE RELATED SERVICES AND PRODUCTS AND YOUR USE THEREOF, including, without limitation, the implied warranties of merchantability, fitness for a particular purpose, and non-infringement. This design is provided on an as-is, where-is, and as-available basis. Kor a Labs do not warrant, endorse, guarantee, or assume responsibility for any product or service advertised or offered by any third party through the product, any open source or third-party software, code, libraries, materials, or information linked to, called by, referenced by or

accessible through the report, its content, and the related services, assets and products, any hyperlinked websites, any websites or mobile applications appearing on any advertising of any of the projects mentioned in this design.

THIS REPORT SHOULD NOT BE USED IN ANY WAY BY ANYONE TO MAKE DECISIONS AROUND INVESTMENT OR INVOLVEMENT WITH ANY PARTICULAR PROJECT, services or assets, especially not to make decisions to buy or sell any assets or products. This report provides general information and is not tailored to anyone's specific situation, its content, access, and/or usage thereof, including any associated services or materials, shall not be considered or relied upon as any form of financial, investment, tax, legal, regulatory, or other advice. This report is based on the scope of materials and documentation provided for a limited review at the time provided. The Kora Labs prepared this design document as an informational exercise documenting the due diligence involved in the course of development of the Kora Labs' software only, and THIS REPORT MAKES NO CLAIMS OR GUARANTEES CONCERNING THE SOFTWARE OPERATION ON DEPLOYMENT OR POST-DEPLOYMENT. This report provides no opinion or guarantee on the security of the code, smart contracts, project, the related assets or anything else at the time of deployment or post deployment.

THE INFORMATION CONTAINED IN THIS REPORT MAY NOT BE COMPLETE NOR INCLUSIVE OF ALL VULNERABILITIES OR DESIGN FLAWS. This report is not comprehensive in scope, it excludes a number of components critical to the correct operation of this system. You agree that your access to and/or use of, including but not limited to, any associated services, products, protocols, platforms, content, assets, and materials will be at your sole risk. On its own, it cannot be considered a sufficient assessment of the correctness of the code or any technology. This design document represents an extensive assessing process intending to help Kora Labs increase the quality of their code while reducing the high level of risk presented by cryptographic tokens, peer-to-peer communications, and blockchain technology, however blockchain technology, peer-to-peer communications, and cryptographic assets present a high level of ongoing risk, including but not limited to unknown risks and flaws.