Mixnet

This repository contains implementation of a mix network, based on the paper Untraceable electronic mail, return addresses, and digital pseudonyms.

System Architecture and Design Decisions

The code is provided as a python package named `mixnet`. It is built from two main components: `mixserver` and `client`.

Each component exposes its API via gRPC methods. The full API can be found in `protos/mixnet.proto`.

The code uses asynchronous programming with `asyncio` and `grpc.aio` to handle multiple requests concurrently.

Run Flow

1. `config.yaml` is placed in a shared config directory.
2. The configuration files containes:
    - Number of messages required per round
    - Round duration
    - Dummy payload
    - List of mix servers and their addresses
    - List of clients and their addresses
3. 3 mix servers are deployed, each writing its own public key to the config directory.
4. Clients are deployed, each writing its own public key to the config directory.
5. Each client registers to the first mix server in the list.
6. Each client calls `WaitForStart` on the first mix server and waits for a respnse.
7. Once all the clients are registered, the first mix server returns a `WaitForStartResponse` with the round duration.
8. Each client starts counting rounds and sending a message each round.
9. Each client exposes `PrepareMessage` gRPC method to prepare a message for sending.
10. At the end of each round, each client sends the prepared messages to the first mix server.
11. If it does not have one, it prepares a "dummy" message contains the dummy payload and sends it.
12. Once the first mix server receives all the messages for the current round, it forwards them to the second mix server.
13. The second mix server does the same and forwards the messages to the third mix server.
14. The third mix server indicates that the message's destination address is a client address, and stores it.
15. When a client wants to receive messages, it calls `PollMessages` on the third mix server, and receives list of messages that are intended to it.

Core Components

MixServer

- **Role in Mixnet**: The `MixServer` acts as a core node in the mix network, responsible for receiving, processing, and forwarding messages between clients and other mix servers, ensuring anonymity and unlinkability.
- **gRPC-Based Communication**: All interactions (registration, message forwarding, polling, and round synchronization) are implemented as asynchronous gRPC service methods, supporting scalable and robust network communication.
- **Client Registration and Synchronization**: Clients must register with the server before participating. The server tracks registered clients and only starts a round when the required number of clients is reached, using an `asyncio.Event` for synchronization.
- **Message Handling and Forwarding**:
    - Messages are received, decrypted, and validated.
    - Messages are stored per round and processed only when all expected messages for a round are received.
    - If a message is destined for a local client, it is stored and written to disk; if for another server, it is forwarded via gRPC.
- **Round-Based Processing**: The server operates in discrete rounds, collecting a fixed number of messages per round. Processing and forwarding are triggered only when all messages for the round are present, ensuring batch anonymity.
- **Concurrency and Synchronization**: Uses `asyncio.Condition` and background tasks to coordinate message collection and processing, allowing non-blocking, concurrent operations.
- **Metrics and Observability**: Optional metrics collection (e.g., round start/end times) is supported for benchmarking and analysis, aiding in performance evaluation.
- **Key Management**: Each server generates and manages its own public/private key pair for message decryption and authentication, with keys stored in a configurable directory.
- **Graceful Shutdown**: The server supports clean shutdown, ensuring all background tasks are completed and resources (such as keys) are cleaned up.

Cleint

- **Role in Mixnet**: The `Client` acts as an endpoint in the mix network, responsible for preparing, sending, and receiving messages through the mix servers, ensuring sender and receiver anonymity.
- **gRPC-Based Communication**: The client exposes its API via asynchronous gRPC methods for message preparation and polling, and interacts with mix servers using gRPC for registration, synchronization, and message forwarding.
- **Onion Encryption**: Messages are encrypted in multiple layers (onion encryption), first with the recipient's public key, then with each mix server's public key in reverse order, ensuring that only the intended recipient can fully decrypt the message.
- **Round-Based Messaging**: The client operates in rounds, preparing and sending one message per round. If no real message is available, a dummy message is sent to maintain traffic consistency and

anonymity.

- **Registration and Synchronization**: Each client registers with the first mix server and waits for a signal to start, ensuring all clients begin sending messages simultaneously for each round.
- **Polling and Decryption**: Clients poll the last mix server for messages intended for them, decrypting each message and filtering out dummy payloads to retrieve only real messages.
- **Concurrency and Asynchronous Operations**: Uses `asyncio` and background tasks to handle message preparation, sending, and polling concurrently, supporting scalable and responsive client behavior.
- **Metrics and Observability**: Optional metrics collection (e.g., message preparation times) is supported for benchmarking and analysis, aiding in performance evaluation.
- **Key Management**: Each client generates and manages its own public/private key pair for message encryption and decryption, with keys stored in a configurable directory.
- **Graceful Shutdown**: The client supports clean shutdown, ensuring all background tasks are completed and resources (such as keys) are cleaned up.

Security Analysis

Threat Model

Mix networks are designed to protect user privacy against a powerful adversary who can observe all network traffic, including message timings, sizes, and routes. The standard threat model assumes:

- **Global Passive Adversary**: Can monitor all network communications and attempt to correlate senders and receivers.
- **Active Adversary**: May inject, drop, or modify messages, and potentially compromise some mix nodes.
- **No Trust in Network Nodes**: Except for cryptographic guarantees, nodes may be malicious or colluding.

The adversary's goal is to detect anomalies in communication by linking senders to receivers or correlating messages across the network.

Anonymity Guarantees

The system provides strong sender and receiver anonymity through several mechanisms:

- **Onion Routing (Layered Encryption)**: Each message is encrypted in multiple layers—first with the recipient's public key, then with each mix server's public key in reverse order. As a message traverses the mix network, each server peels off one layer, learning only where to forward the message next, but not the original sender or final recipient.
- **Batch Processing and Dummy Messages**: Messages are processed in rounds, and if a client has no real message, it sends a dummy message. This prevents traffic analysis by ensuring a constant message rate and hiding real communication patterns.
- **Address and Payload Confidentiality**: Only the intended recipient can fully decrypt the message and learn its contents and destination.

This design ensures that:

- **Sender Anonymity**: Mix servers cannot link incoming messages to outgoing ones, and the recipient cannot identify the sender.
- **Receiver Anonymity**: Observers and mix servers cannot determine the final recipient until the last layer is decrypted.

Layered Encryption and Nonces

Layered encryption (onion encryption) is central to mixnet security:

- **Each Layer Protects Routing Information**: Only the mix server with the corresponding private key can decrypt its layer and learn where to forward the message next.
- **End-to-End Confidentiality**: The innermost layer is encrypted with the recipient's public key, so only the recipient can read the message.
- **Prevents Linkability**: Even if an adversary controls some mix servers, they cannot correlate input and output messages without breaking the encryption.

Random values (nonces) are included in encrypted messages to:

- **Ensure Semantic Security**: Nonces prevent identical plaintexts from producing identical ciphertexts, thwarting replay and correlation attacks.
- **Prevent Replay Attacks**: Each message is unique, so replayed messages are easily detected and discarded.
- **Increase Unpredictability**: Nonces add randomness, making it harder for adversaries to analyze or guess message contents.

Cryptographic Components Used

The implementation uses the following cryptographic primitives (see `src/mixnet/crypto.py`):

- **NaCl (libsodium) Public-Key Cryptography**: For key generation, encryption, and decryption.
  - `PrivateKey` and `PublicKey`: Used to generate and manage key pairs for clients and servers.
  - `SealedBox`: Provides anonymous public-key encryption, allowing messages to be encrypted for a recipient without revealing the sender.
- **Base64 Encoding**: Keys are serialized and stored in Base64 format for portability.
- **Layered Encryption**: The `encrypt` function applies public-key encryption for each mix server and the recipient, forming the onion layers.

Benchmarks

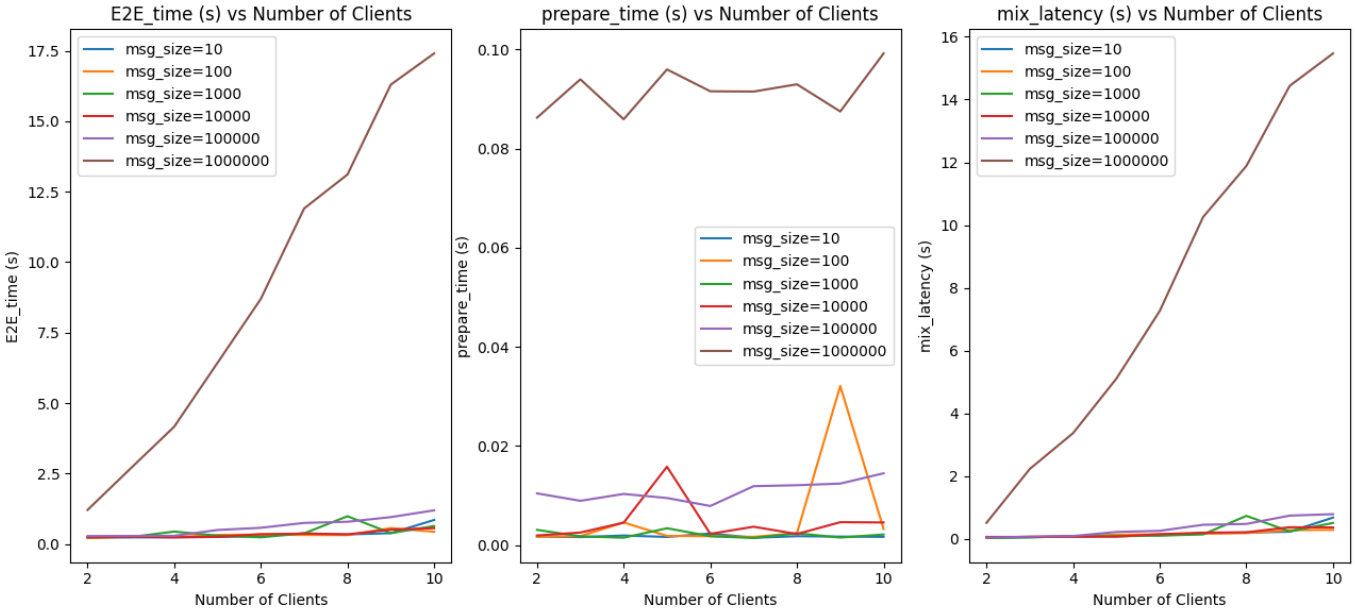
In `src/mixnet/benchmarks.py`, I provide a benchmark for measuring three metrics:

1. Message preparation time (client side)
2. Decryption and forwarding latency (mix side)
3. End-to-end delivery time

I ran the benchmark with 2 to 10 clients, and with message size from 10 to 10^6 bytes. In each run, I explicitly sent a message from client_1 to client_2, while all the other clients sent to themselves.

I measured the metrics on this specific message round.

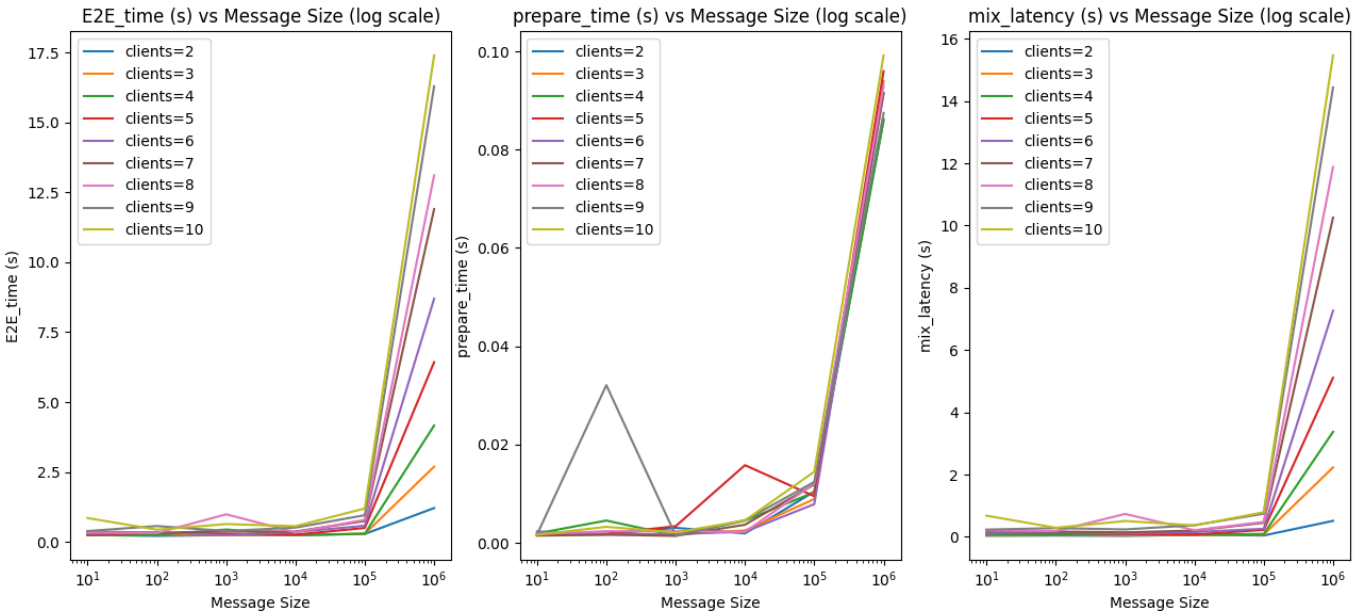
I received the following results:



For end-to-end delivery time and mixes latency, we can see that it is linear with the number of clients, since each client sends a message in each round.

For message preparation time, we can see that the number of clients does not affect it, since each client prepares its message independently.

We can see though that the message size has exponential effect on the metrics, since each message is encrypted with multiple layers of public-key encryption, and the encryption time grows with the message size. Also delivery time grows with the message size, since each mix server needs to decrypt the message and forward it to the next server.



Here the message size axis is logarithmic. We can see that the message size has exponential effect on the metrics, as mentioned above.

We can also see that the number of clients has linear effect on the metrics, as we saw above.

Proposed Attacks

Below are two plausible attack scenarios that could compromise the anonymity or integrity of the mix network, as described in the assignment, along with their respective mitigations. Each scenario considers adversaries capable of observing or interacting with the system.

Attack Scenario 1: Traffic Analysis

- **Description**: An adversary observes the timing and volume of messages entering and exiting the mix network across its three mix servers. By correlating these patterns, particularly if messages are not sufficiently delayed or batched, the adversary may link senders to recipients, undermining anonymity.
- **Mitigation**: Implement batching and random delays to process messages in fixed-size batches per round, making input-output correlation more difficult. Additionally, pad messages to uniform sizes and use dummy traffic to obscure real message patterns.

Attack Scenario 2: Compromised Mix Server

- **Description**: If one of the three mix servers is compromised, an adversary could access its decryption keys, observe message contents, or manipulate them during decryption and forwarding, compromising both anonymity and integrity.
- **Mitigation**: Use threshold cryptography to distribute decryption keys across mix servers, requiring multiple servers to collaborate for decryption, ensuring a single compromise is insufficient to break the system.