## 1. System Architecture Design

Client-Server Model:

Server: Acts as the CA, handling key registration, certificate issuance, and possibly certificate revocation. This server will maintain a database or a secure store of public keys and corresponding certificates.

Clients: Users (like Alice and Bob) who will register with the CA to obtain their certificates and then use these certificates to authenticate themselves and encrypt communications with other clients.

## 2. Server as a Certificate Authority

Key and Certificate Management:

Registration and Key Submission: Clients generate their key pairs locally and submit their public keys to the server for certification.

Certificate Issuance: The server signs the public keys with its own private key, creating certificates which are then sent back to the clients.

Certificate Storage: The server maintains a list or database of all issued certificates and their corresponding public keys, which can be referenced for authentication and verification during communications.

## 3. Client Implementation

Initial Setup:

Key Pair Generation: Each client generates an RSA key pair upon their initial setup.

Register with the CA: Submit the public key to the server and receive a certificate.

Communication with Other Clients:

Certificate Exchange: Before secure communication can start, clients exchange certificates.

Verification: Clients use the CA's public key (which should be pre-installed or securely obtained) to verify the certificates received from other clients.

Secure Messaging: Use the public key from the verified certificate to encrypt messages and establish secure channels (e.g., using AES for symmetric encryption of session communications).

## 4. Secure Communication Protocols

Using TLS/SSL Concepts:

Employ concepts similar to TLS/SSL for secure communications:

Handshake Protocol: Implement a handshake mechanism where certificates are exchanged and verified before any sensitive data is transmitted.

Session Keys: Use techniques like Diffie-Hellman within this secure environment to generate session keys for symmetric encryption.

## 5. Scalability and Security Measures

Scalability: The server should be designed to handle multiple connections simultaneously and manage a growing number of client registrations and certificate issuances.

Security Features: Implement logging, monitoring, and possibly rate limiting to prevent abuse (e.g., denial-of-service attacks) and to enhance security.

## 6. Testing and Documentation

Testing: Rigorously test the registration process, certificate issuance, and secure communication. Include tests for error handling and security breach attempts.

Documentation: Provide comprehensive documentation covering the architecture, flow of data, security measures, and user instructions.

## 7. Technology and Tools

Python Libraries: Utilize libraries like Flask for setting up the server, SQLAlchemy for database interactions, and PyCryptodome for cryptographic functions.

Networking: Use secure protocols and ensure all data transmitted between the client and server is encrypted.

By structuring your system in this way, you ensure a scalable, secure, and efficient CA model that supports the dynamic registration of new users while providing robust security features for communications. This setup not only fulfills the assignment's requirements but also offers a realistic model for secure network communication systems.

Flow:

Receiving message and Yusuf is Registered:

**CA and server are used interchangeably**

1. Yusuf connects to server
2. Y signs in (already registered with CA)
3. Server prompts directory for messages for Y
4. Y downloads messages meant for him
   1. Automatically send certificate for sender, gets sent certificate encrypted with CA private key from CA
      1. Good for man in the middle attacks
   2. (For each message, session key included)
   3. Decode from Base64 to utf-8
   4. Decrypt session key component with Y Private Key
   5. Decrypt message body with session key (gotten from prev step)
   6. Decompress message body (to get actual message and signature)
   7. Decrypt message digest with Greg’s (senders) Public Key
   8. Verify Message is unchanged (check integrity) compare hashes

Sending Message and Greg is Registered:

1. Greg connects to server
2. G signs in
3. G wants to send message to Y
   1. Query server to see if Y is registered (the following assumes Y is registered)
4. CA sends G a certificate that is encrypted with CA’s private key
   1. Decrypt with CA public key
   2. Certificate includes Yusuf’s Public key
5. Encrypt session key with Y public key
6. Encrypt message digest with G private key
7. Compress message body (everything except session key)
8. Encrypt zipped file with session key
9. Prepend to encrypted message
10. Encode and send message to Server

Non-registered User:

1. Prompted to register or exit

Server:

* Acts as a
  + Certificate Authority (CA)
  + Authentication Server (AS)
  + Message store
  + Communication bridge

Todo:

* Priority:
* Integrate CA into server
  + Directory/ data store
    - Possibly JSON
* What the hell DH?? Ask bossman about symmetric encryption
* Hashing
* Compression
* Encoding
* Message format
* Implement image + caption sending

NOTES:  
- Currently, email acts as the keyID

A diagram of a process

Description automatically generated

**ASSIGNMENT BRIEF REQUIREMENTS**

Cryptosystem Design

From the assignment brief, the system design mimics Pretty Good Privacy (PGP) by combining multiple cryptographic techniques:

1. Asymmetric Encryption (RSA) for encrypting messages and verifying identity.

2. Symmetric Encryption (AES) for faster, secure message content encryption.

3. Certificate-Based Authentication for ensuring the authenticity of public keys via a trusted Certification Authority (CA).

4. Hashing for message integrity and non-repudiation.

Communication Connectivity Model

The connectivity model described suggests using TCP for a robust, ordered, and error-checked delivery of streams of bytes. In this model:

- Both clients (Alice and Bob) can act as senders and receivers.

- The connection is established directly between the two clients, likely using Python's `socket` library for network communication.

Key Management

Key management, as per the assignment, involves:

1. Private and Public Key Pairs: Each party has their own private key and shares the public key.

2. Certificates: Public keys are distributed along with certificates signed by the CA, which are used to verify the authenticity of the key holders.

3. Shared Secret Keys: Using protocols like Diffie-Hellman for securely generating and sharing symmetric keys for session encryption.

Choice of Cryptographic Algorithms

1. RSA for Public-Key Cryptography: Used for encrypting the keys and digital signatures. RSA is chosen for its security and wide adoption.

2. AES for Symmetric Key Cryptography: Utilized for encrypting message content due to its efficiency in processing large amounts of data securely.

3. PKCS1 and PKCS5 Padding: These padding schemes are used to ensure that block cipher encryption stays secure by making the input size fit the required block size.

**OUR SYSTEMS**

**CLIENT.py**

Cryptosystem Design

1. Asymmetric Key Generation: The script uses a function to generate RSA key pairs based on user-provided data (like email and passphrase). This key pair is used for asymmetric operations such as encryption and signing.

2. Symmetric Key Handling: Session keys for symmetric encryption (AES) are dynamically generated using the AES CTR (Counter) mode, showcasing how the system deals with session-level encryption separately from the persistent RSA keys.

3. Encrypted Communication: Data, such as nonces (number used once) and session keys, are encrypted and decrypted using the respective user's RSA keys or symmetric keys, facilitating a secure channel for sensitive operations.

Communication Connectivity Model

1. Socket Programming: The code utilizes Python's `socket` library to establish TCP connections, allowing for reliable, ordered, and error-checked transmission of data between the client and the server.

2. Message Exchange: The client initiates communication by connecting to the server's socket, followed by a series of request-response messages that handle login, key exchange, and further secure communications.

Key Management

1. Private/Public Keys and Certificate Handling: The client manages its RSA keys and also interacts with the server to validate and exchange public keys/certificates, ensuring the authenticity of communication parties.

2. Secure Storage and Retrieval: Keys are generated, possibly stored, and retrieved securely, incorporating user input (like passphrase) to access private keys.

Choice of Cryptographic Algorithms

1. RSA: Utilized for secure key exchange, digital signatures, and encrypting smaller pieces of data like session keys.

2. AES-CTR: Chosen for encrypting message contents, which might include larger data sizes typical in file transfers or image sharing. The CTR mode is used for its advantages in creating a stream cipher from a block cipher without the need for padding.

**SERVER.py**

Now, let's analyze the `server.py` file to better understand the server-side mechanisms that interact with this client setup. This will help in comprehensively describing the full cryptosystem design, especially focusing on how the server handles the key management and cryptographic operations detailed in the client code.

The `server.py` code is structured to handle communication and cryptographic tasks in coordination with the client-side script (`client.py`). Here's an outline of the key components and how they contribute to the overall system:

Cryptosystem Design

1. Handling Cryptographic Requests: The server manages requests for sending and receiving encrypted data. This includes utilizing the GNU Privacy Guard (GnuPG) for cryptographic functions, reflecting a robust setup for dealing with encryption and decryption.

2. GnuPG Integration: The use of GnuPG is notable for handling cryptographic operations, including key management and message encryption/decryption.

Communication Connectivity Model

1. TCP Server Setup: The server script uses Python’s `socket` library to set up a TCP server that listens for incoming connections, ensuring a reliable channel for data transmission.

2. Multithreading for Client Handling: The server employs threading to manage multiple client connections simultaneously, allowing it to handle various client requests without blocking.

Key Management

1. Certificate and Key Management: The server likely handles the verification of certificates received from clients and manages the distribution of its public key and the CA's public key to clients upon request.

2. Registration and Login Management: The server provides functionality for user registration and login, which includes storing and retrieving user-specific keys and handling authentication based on received credentials.

Choice of Cryptographic Algorithms

1. Public and Private Key Operations: Similar to the client, the server uses RSA for handling public/private key operations, reinforcing the use of secure, established algorithms for cryptographic tasks.

2. Symmetric Encryption for Session Data: While the detailed implementation in the truncated output isn’t fully visible, the server is designed to handle symmetric key encryption for session data, likely using AES, consistent with the client implementation.

Implementation Details

- User Authentication and Command Handling: The script handles basic commands like SEND, RECEIVE, LOGIN, SIGN UP, and QUIT, allowing users to interact with the server through a simple text-based menu.

- Secure Message Format: Messages are expected to be in a structured format, including elements like sender, recipient, timestamp, sender's public key, CA's signature, and the encrypted message itself, emphasizing a structured approach to secure communications.

Overall, the combination of the `client.py` and `server.py` scripts outlines a comprehensive system for secure communication between clients, managed by a server that implements cryptographic best practices and robust network programming techniques. This system reflects the detailed requirements of the assignment brief, focusing on security, reliability, and effective communication management.