Analisi Tecnica Dettagliata di un Server UDP con Crittografia Asimmetrica e Gestione della Frammentazione: `server_UDP_crypto_protocollo.py`

Giovanni Viva

11 maggio 2025

Sommario

Il presente articolo offre un'analisi tecnica approfondita dello script Python `server_UDP_crypto_protocollo.py`, un server UDP progettato per comunicazioni sicure. Questo server implementa la crittografia asimmetrica RSA per lo scambio di chiavi e la cifratura dei messaggi, insieme a un meccanismo robusto per la frammentazione e il riassemblaggio di datagrammi UDP di grandi dimensioni. L'analisi esplora l'architettura del server, il suo flusso operativo, i protocolli impiegati, lo scopo primario, le intenzioni didattiche dell'autore e potenziali casi d'uso. Vengono inoltre discussi i punti di forza e le possibili aree di miglioramento, fornendo una valutazione complessiva della soluzione.

Indice

1 Introduzione **2**

2 Analisi del Funzionamento (`Come funziona il codice server_UDP_crypto_protocollo.py`) **2**

2.1 Architettura Generale 2
2.2 Flusso di Esecuzione Principale 2
2.3 Componenti Chiave 3
2.4 Interazioni 5
2.5 Protocolli e Meccanismi 6
2.6 Librerie Utilizzate 7

3 Scopo Primario del Codice **7**

4 Intenzione Didattica e Comunicativa dell'Autore **8**

5 Esempio d'Uso **9**

6 Conclusioni **10**

7 File di Riferimento **11**

1 Introduzione

Nel panorama delle comunicazioni di rete, il User Datagram Protocol (UDP) è apprezzato per la sua leggerezza e bassa latenza, rendendolo ideale per applicazioni sensibili al tempo come lo streaming video, i giochi online e i sistemi di telemetria. Tuttavia, la sua natura connectionless e la mancanza di meccanismi intrinseci di affidabilità e sicurezza pongono sfide significative. UDP non garantisce la consegna dei pacchetti, né il loro ordine, né la protezione contro intercettazioni o manomissioni.

Lo script Python `server_UDP_crypto_protocollo.py` si propone come una soluzione per mitigare alcune di queste limitazioni, implementando un server UDP che integra funzionalità di crittografia asimmetrica e un sistema di gestione della frammentazione dei messaggi. Questo approccio ibrido mira a combinare la velocità di UDP con un livello di sicurezza e affidabilità sufficiente per determinate applicazioni.

L'obiettivo di questo articolo è dissezionare il funzionamento interno di
`server_UDP_crypto_protocollo.py`,
analizzando le tecniche di programmazione, i protocolli di comunicazione e le scelte architetturali adottate. Esploreremo come il server gestisce lo scambio di chiavi pubbliche, la cifratura e decifratura dei dati, e il complesso processo di frammentazione e riassemblaggio dei messaggi, fondamentale quando si superano le dimensioni massime dei datagrammi UDP.

2 Analisi del Funzionamento (`Come funziona il codice server_UDP_crypto_protocollo.py`)

2.1 Architettura Generale

Il server è implementato come una singola classe, `UDPServer`, che incapsula tutta la logica di gestione della rete, della crittografia e della frammentazione. L'architettura è multithreading: un thread principale gestisce l'avvio e l'arresto del server, mentre un thread dedicato, `receiver_thread`, è responsabile dell'ascolto e della ricezione dei pacchetti UDP. Questa separazione previene il blocco del thread principale durante le operazioni di I/O di rete.

La gestione dei dati condivisi tra thread, come l'oggetto socket e i buffer dei frammenti, è protetta mediante un `threading.Lock` (`_socket_lock`), garantendo l'accesso mutuamente esclusivo e prevenendo race condition. Un `threading.Event` (`_stop_event`) viene utilizzato per segnalare al thread ricevitore la necessità di terminare in modo pulito.

2.2 Flusso di Esecuzione Principale

L'esecuzione del server inizia nella funzione `main()`:

1. Viene istanziato un oggetto `UDPServer`, specificando host, porta e timeout per i frammenti.
2. Viene opzionalmente assegnata una funzione di callback (`on_message`) che sarà invocata al ricevimento di un messaggio completo e decifrato.
3. Il metodo `start()` del server viene chiamato.

4. Il thread principale entra in un loop di attesa, mantenendo il server attivo fino a un'interruzione da tastiera (`KeyboardInterrupt`).

Il metodo `start()` esegue i seguenti passaggi cruciali:

1. Verifica che il server non sia già attivo.
2. Genera una coppia di chiavi RSA (privata e pubblica) a 4096 bit. La chiave pubblica viene serializzata in formato PEM per la trasmissione ai client.
3. Crea un socket UDP (`socket.AF_INET, socket.SOCK_DGRAM`).
4. Imposta un timeout per le operazioni sul socket.
5. Esegue il bind del socket all'indirizzo e alla porta specificati.
6. Imposta lo stato del server su attivo (`is_active = True`).
7. Avvia il `receiver_thread`, che eseguirà il metodo `_receive_loop`.

2.3 Componenti Chiave

- **`UDPServer`**: La classe principale che orchestra tutte le operazioni. Mantiene lo stato del server, gestisce le chiavi crittografiche, i buffer dei frammenti e le chiavi pubbliche dei client.
- **`_receive_loop()`**: Il cuore del server. In un ciclo continuo (finché `_stop_event` non è impostato):
  - Utilizza `select.select()` per attendere dati in arrivo sul socket UDP in modo non bloccante, rispettando il `socket_timeout`.
  - Se dati sono disponibili, li legge con `udp_socket.recvfrom()`.
  - Passa i dati e l'indirizzo del client al metodo `_process_message()` per l'elaborazione.
  - Periodicamente (quando `select` va in timeout senza dati) chiama `_cleanup_fragment_buffers()` per rimuovere buffer di frammenti scaduti.
- **`_process_message(client_address, data)`**: Questo metodo è il primo punto di smistamento per i dati ricevuti. Decide come interpretare il pacchetto:
  1. Se il pacchetto contiene una chiave pubblica PEM (identificata dai marker `---BEGIN PUBLIC KEY---` e `---END PUBLIC KEY---`), la deserializza e la memorizza associata all'indirizzo del client in `self.client_public_keys`.
  2. Se il pacchetto è una richiesta della chiave pubblica del server (`b"REQ_PUB_KEY\n"`), invia la propria chiave pubblica PEM al client.
  3. Se il pacchetto ha una lunghezza minima di 6 byte, tenta di interpretarlo come un frammento di messaggio. Estrae l'ID del messaggio (4 byte), il numero del frammento (1 byte), il numero totale di frammenti (1 byte) e il payload. Se questi valori sono validi, passa i dati a `_process_fragment()`.

4. Altrimenti, tratta il pacchetto come un messaggio non frammentato (presumibilmente cifrato) e lo passa a `_process_non_fragmented_message()`.

- **`_process_fragment(client_address, message_id, fragment_number, total_fragment_count, fragment_payload)`**: Gestisce i frammenti:
  - Inizializza un buffer per il `message_id` specifico di quel `client_address`, se non esiste già.
  - Memorizza il `fragment_payload` nel buffer, indicizzato dal `fragment_number`.
  - Se tutti i frammenti attesi (`total_fragment_count`) sono stati ricevuti:
    * Riassembla i frammenti nell'ordine corretto.
    * Tenta di decifrare il messaggio riassemblato usando `_decrypt_message()`.
    * Se la decifratura ha successo, invoca la callback `on_message` (se definita) e invia un ACK al client (formato: `"message_id:  ACK"`).
    * Rimuove il buffer del messaggio completato.
  - Se non tutti i frammenti sono arrivati, resetta un timer di timeout per quel `message_id` (`_reset_fragment_timer()`).
- **`_reset_fragment_timer(client_address, message_id)`**: Avvia o riavvia un `threading.Timer` per il buffer di un messaggio frammentato. Se il timer scade prima che tutti i frammenti siano ricevuti, il metodo `_timeout_fragment_buffer()` viene chiamato.
- **`_timeout_fragment_buffer(client_address, message_id)`**: Rimuove un buffer di frammenti incompleto quando il suo timer scade, prevenendo l'accumulo di dati parziali.
- **`_cleanup_fragment_buffers()`**: Scansiona periodicamente tutti i buffer di frammenti e rimuove quelli i cui timer sono scaduti e non sono più attivi.
- **`_process_non_fragmented_message(client_address, data)`**: Tenta di decifrare i dati ricevuti come un unico messaggio. Se la decifratura ha successo, invia un semplice `b"ACK"` al client.
- **`_decrypt_message(client_address, message_bytes)`**: Decifra i `message_bytes` utilizzando la chiave privata del server. Questo implica che il messaggio originale sia stato cifrato dal client con la chiave pubblica del server. La decifratura utilizza RSA con padding OAEP e hashing SHA256.
- **`send_to(message, client_address)`**: Invia un messaggio a un client. Se il messaggio non è una richiesta di chiave pubblica o la chiave pubblica stessa, e se la chiave pubblica del client destinatario è nota, il messaggio viene cifrato usando la chiave pubblica del client (RSA-OAEP, SHA256) prima dell'invio. La dimensione massima del frammento per i messaggi in uscita è determinata da `self.max_fragment_size`, anche se la frammentazione in uscita non è esplicitamente implementata in questa versione del metodo `send_to`.
- **`close()`**: Arresta il server in modo pulito, impostando `_stop_event`, chiudendo il socket e attendendo la terminazione del `receiver_thread`.

2.4 Interazioni

Le interazioni tra i componenti sono fondamentali per il funzionamento del server:

- **Client-Server (Scambio Chiavi):**
  1. Il client può inviare la sua chiave pubblica al server. Il server la memorizza.
  2. Il client può richiedere la chiave pubblica del server inviando `REQ_PUB_KEY\n`. Il server risponde con la sua chiave pubblica PEM.
- **Client-Server (Invio Messaggio Cifrato):**
  1. Il client, avendo la chiave pubblica del server, cifra un messaggio.
  2. Se il messaggio è grande, il client lo frammenta, anteponendo a ciascun frammento un header (ID messaggio, num. frammento, tot. frammenti).
  3. Il client invia i frammenti (o il messaggio intero se piccolo) al server.
- **Server (Ricezione e Decifratura):**
  1. Il server riceve i pacchetti.
  2. `_process_message` smista il traffico.
  3. Se frammentato, `_process_fragment` raccoglie i pezzi.
  4. Una volta assemblato (o se non frammentato), `_decrypt_message` usa la chiave privata del server per decifrare.
  5. Viene inviato un ACK al client.
- **Server-Client (Invio Risposta Cifrata):**
  1. Il server, per inviare una risposta (es. ACK), usa `send_to`.
  2. Se la chiave pubblica del client è nota, cifra il messaggio con essa.
- **Gestione Concorrenza:**
  - `_socket_lock` protegge l'accesso al socket condiviso `self.udp_socket`.
  - I buffer dei frammenti (`client_message_fragment_buffers`) sono strutture dati condivise, ma le operazioni su di essi sono gestite principalmente dal thread ricevitore, mitigando alcuni rischi di concorrenza diretta. Tuttavia, la loro modifica (aggiunta/-rimozione di frammenti e buffer interi) deve essere considerata thread-safe se più thread potessero accedervi, sebbene qui il design limiti tale accesso al `receiver_thread` per le modifiche principali. I timer di frammentazione (`threading.Timer`) operano in thread separati e le loro callback (`_timeout_fragment_buffer`) modificano i buffer, quindi la struttura dei buffer deve implicitamente gestire questa concorrenza (es. Python dicts sono thread-safe per operazioni atomiche come `del`).

2.5 Protocolli e Meccanismi

- **UDP:** Protocollo di trasporto a livello 4, connectionless.
- **Protocollo di Frammentazione Custom:** Il server si aspetta che i messaggi frammentati dai client seguano un formato specifico:
  - `Message ID (4 byte, big-endian)`: Identificatore univoco per un messaggio completo, generato dal client.
  - `Fragment Number (1 byte)`: Numero progressivo del frammento, a partire da 1.
  - `Total Fragment Count (1 byte)`: Numero totale di frammenti per quel messaggio.
  - `Payload (restanti byte)`: Il contenuto effettivo del frammento.

  La dimensione massima del payload di un frammento in entrata è implicitamente limitata da `self.buffer_size` (dimensione del buffer di ricezione del socket) meno i 6 byte dell'header. La dimensione `max_fragment_size = 490` è un parametro per la frammentazione in uscita (sebbene non implementata), suggerendo un payload massimo per frammento di 490 byte per evitare la frammentazione IP (MTU tipica Ethernet è 1500 byte, meno header IP e UDP).
- **Scambio Chiavi Asimmetriche:**
  - Il server genera una coppia RSA a 4096 bit.
  - I client inviano la loro chiave pubblica in formato PEM.
  - Il server invia la sua chiave pubblica PEM su richiesta (`REQ_PUB_KEY\n`).
- **Crittografia Asimmetrica (RSA):**
  - Utilizzata per cifrare/decifrare i messaggi. Un messaggio inviato al server viene cifrato dal client con la chiave pubblica del server. Un messaggio inviato dal server a un client viene cifrato dal server con la chiave pubblica di quel client.
  - Schema di padding: OAEP (Optimal Asymmetric Encryption Padding).
  - Funzione di hash per MGF1 (Mask Generation Function) e per l'algoritmo di OAEP: SHA256.
- **Gestione Timeout:**
  - `socket_timeout`: Timeout per le operazioni di ricezione sul socket UDP.
  - `fragment_timeout`: Timeout per il riassemblaggio dei frammenti. Se un messaggio non viene completato entro questo tempo, i suoi frammenti vengono scartati.
- **Logging:** Utilizzo del modulo `logging` per tracciare le operazioni del server, avvisi ed errori.
- **ACK dei Messaggi:** Dopo aver riassemblato e decifrato con successo un messaggio frammentato, il server invia un ACK al client nel formato `"<message_id>: ACK"`. Per i messaggi non frammentati, invia un semplice `b"ACK"`.

2.6 Librerie Utilizzate

- `socket`: Per la comunicazione di rete UDP di basso livello (creazione socket, bind, invio, ricezione).
- `select`: Per il multiplexing I/O, in particolare `select.select()` per attendere dati sul socket in modo non bloccante.
- `logging`: Per la registrazione di eventi, debug e messaggi di errore.
- `threading`: Per la programmazione concorrente (creazione del thread ricevitore, lock per la sincronizzazione, event per la segnalazione, timer per i timeout dei frammenti).
- `sys`: Per l'accesso a parametri specifici del sistema, come `sys.stdout` per l'handler del logger.
- `errno`: Per interpretare i codici di errore delle operazioni di sistema/socket (es. `ECONNREFUSED`).
- `time`: Per funzioni correlate al tempo, come `time.sleep()`.
- `os`: Utilizzato nel commento per la generazione di ID messaggio unici, ma non attivamente nel codice fornito per questa funzionalità specifica lato server (gli ID messaggio sono generati dal client).
- `cryptography`: Libreria fondamentale per le operazioni crittografiche.
  - `hazmat.primitives.asymmetric.rsa`: Per la generazione di chiavi RSA (`generate_private_key`) e operazioni di cifratura/decifratura.
  - `hazmat.primitives.asymmetric.padding`: Per specificare lo schema di padding OAEP.
  - `hazmat.primitives.hashes`: Per specificare algoritmi di hash (SHA256).
  - `hazmat.primitives.serialization`: Per serializzare le chiavi pubbliche in formato PEM (`public_bytes`) e deserializzare chiavi PEM ricevute (`load_pem_public_key`).
  - `cryptography.exceptions.InvalidSignature`: Menzionata nell'import, ma non usata attivamente nel codice fornito (più pertinente per la verifica di firme digitali, che non è implementata qui).
  - `hazmat.backends.default_backend`: Per selezionare il backend crittografico predefinito.

3 Scopo Primario del Codice

Lo scopo primario di `server_UDP_crypto_protocollo.py` è fornire una piattaforma server per comunicazioni basate su UDP che siano:

1. **Sicure:** Attraverso l'uso della crittografia asimmetrica RSA, il server mira a garantire la confidenzialità dei dati scambiati. I messaggi sono cifrati in modo che solo il destinatario designato (che possiede la chiave privata corrispondente) possa decifrarli.

2. **Capaci di gestire messaggi di grandi dimensioni:** UDP ha una dimensione massima del payload per datagramma. Per superare questa limitazione, il server implementa un meccanismo di ricezione e riassemblaggio di messaggi frammentati, aspettandosi che i client inviino dati più grandi suddivisi secondo un protocollo di frammentazione custom.

3. **Robuste contro la perdita parziale di messaggi frammentati:** Grazie ai timeout sui buffer dei frammenti, il server evita di conservare indefinitamente dati incompleti, liberando risorse.

Il codice cerca di risolvere i seguenti problemi intrinseci o comuni nelle comunicazioni UDP:

- **Mancanza di confidenzialità:** I pacchetti UDP standard sono trasmessi in chiaro. La crittografia RSA affronta questo.
- **Limitazione sulla dimensione dei messaggi (MTU):** La frammentazione permette l'invio logico di messaggi più grandi della Maximum Transmission Unit del percorso di rete.
- **Gestione delle risorse in caso di frammenti persi:** Senza un meccanismo di timeout, i frammenti ricevuti di un messaggio incompleto potrebbero occupare memoria indefinitamente.

Il server è progettato per fungere da endpoint affidabile e sicuro per client che necessitano di inviare dati su UDP con queste caratteristiche aggiuntive.

4 Intenzione Didattica e Comunicativa dell'Autore

Analizzando il codice, emergono diverse intenzioni didattiche e comunicative da parte dell'autore:

- **Dimostrare la crittografia asimmetrica applicata:** Il codice illustra chiaramente come generare coppie di chiavi RSA, serializzare/deserializzare chiavi pubbliche in formato PEM, e utilizzare queste chiavi per cifrare e decifrare messaggi. L'uso di RSA con padding OAEP e SHA256 mostra una pratica moderna e sicura.
- **Insegnare la gestione della frammentazione su UDP:** La necessità di frammentare messaggi su UDP è un problema comune. L'implementazione di un protocollo di frammentazione con ID messaggio, numero di frammento e conteggio totale, insieme alla logica di riassemblaggio e gestione dei timeout, è un eccellente esempio pratico.
- **Illustrare la programmazione di rete multithreaded:** L'uso di un thread separato per la ricezione dei messaggi (`_receive_loop`) è una tecnica standard per server di rete che devono rimanere responsivi. La gestione della sincronizzazione (`Lock`) e della terminazione pulita (`Event`) sono concetti importanti.
- **Enfatizzare la robustezza del server:** L'inclusione di timeout per le operazioni socket e per i buffer dei frammenti, la gestione degli errori (`try-except`, logging), e la pulizia

dei buffer scaduti (`_cleanup_fragment_buffers`) dimostrano un'attenzione alla stabilità e all'uso efficiente delle risorse.

- **Mostrare l'utilizzo della libreria `cryptography`:** Il codice serve come esempio pratico di come utilizzare le API di questa potente libreria Python per compiti crittografici comuni.
- **Presentare un protocollo di comunicazione semplice ma completo:** Lo scambio di chiavi (richiesta/risposta), l'invio di dati cifrati e la ricezione di ACK formano un mini-protocollo. L'ACK con ID messaggio (`id_appg + ":  ACK"`) è un dettaglio che mostra la volontà di confermare la ricezione specifica del messaggio.
- **Chiarezza e leggibilità del codice:** L'uso di nomi di variabili e metodi descrittivi, commenti esplicativi e una struttura di classe logica contribuiscono alla comprensibilità del codice, suggerendo un'intenzione di renderlo accessibile per l'apprendimento.

L'autore sembra voler comunicare non solo *come* implementare queste funzionalità, ma anche *perché* sono importanti in un contesto di comunicazioni UDP sicure e affidabili. La scelta di RSA a 4096 bit, ad esempio, indica una preferenza per una sicurezza elevata, anche se a scapito di una maggiore overhead computazionale rispetto a chiavi più corte o alla crittografia simmetrica (che però richiederebbe uno scambio sicuro di chiavi simmetriche, spesso realizzato proprio con l'asimmetrica).

5 Esempio d'Uso

Un server con le caratteristiche di `server_UDP_crypto_protocollo.py` potrebbe essere utilmente impiegato in diversi scenari:

1. **Sistemi di Telemetria e Monitoraggio Sicuri:** In contesti industriali (IIoT) o infrastrutturali, dove sensori e dispositivi inviano periodicamente dati di stato o misurazioni. UDP è preferito per la bassa overhead, ma la sicurezza è cruciale. La frammentazione permette l'invio di report dettagliati. *Esempio:* Sensori ambientali in una rete geograficamente distribuita inviano log di dati cifrati a un server centrale per l'analisi.
2. **Comunicazione Leggera per Videogiochi Multiplayer:** Alcuni tipi di dati di gioco (es. aggiornamenti di stato rapidi ma non critici per la consistenza assoluta) beneficiano di UDP. La crittografia protegge da cheat o sniffing, e la frammentazione può essere usata per messaggi di gioco più complessi (es. chat di gioco, dati di configurazione iniziale). *Esempio:* Invio di comandi di input del giocatore o aggiornamenti di posizione cifrati.
3. **Sistemi di Notifica Sicuri e Veloci:** Per inviare notifiche push o allerte a client specifici, dove la consegna rapida è importante e la connessione persistente di TCP non è desiderata. La crittografia assicura che solo il destinatario possa leggere la notifica. *Esempio:* Un sistema di allerta invia messaggi cifrati a dispositivi mobili o desktop client.

4. **Trasferimento di File di Piccole/Medie Dimensioni su Reti Inaffidabili:** Sebbene TCP sia generalmente preferito per il trasferimento file, in scenari con alta perdita di pacchetti o dove il setup di una connessione TCP è problematico, un protocollo UDP con frammentazione e crittografia potrebbe essere una valida alternativa per file non eccessivamente grandi, specialmente se si aggiungesse un meccanismo di ritrasmissione selettiva dei frammenti. *Esempio:* Invio di file di configurazione o piccoli aggiornamenti software a dispositivi remoti.

5. **Servizi di Chat Semplici e Sicuri Peer-to-Peer (con server per discovery/relay):** Sebbene il codice sia un server, la logica di crittografia e frammentazione potrebbe essere adattata per comunicazioni dirette tra client, con il server che agisce inizialmente per lo scambio di chiavi pubbliche e l'inoltro di messaggi se il NAT traversal diretto fallisce. *Esempio:* Client di chat che scambiano messaggi cifrati e potenzialmente frammentati.

In tutti questi casi, la combinazione di velocità (UDP), sicurezza (RSA) e capacità di gestire dati di varia dimensione (frammentazione) offerta da questo server rappresenta un compromesso interessante.

6 Conclusioni

L'analisi di `server_UDP_crypto_protocollo.py` rivela un software ben strutturato e concettualmente solido per la creazione di un server UDP con funzionalità avanzate di sicurezza e gestione dei dati.

Punti di Forza:

- **Sicurezza Robusta:** L'adozione di RSA a 4096 bit con padding OAEP e SHA256 offre un elevato livello di confidenzialità per i messaggi scambiati.
- **Gestione della Frammentazione:** Il meccanismo di ricezione e riassemblaggio dei frammenti, completo di timeout, è essenziale per superare le limitazioni di dimensione dei datagrammi UDP e gestire in modo robusto la perdita parziale di messaggi.
- **Architettura Multithreading:** L'uso di un thread dedicato per la ricezione dei pacchetti garantisce che il server rimanga responsivo.
- **Chiarezza del Codice:** Il codice è generalmente ben commentato e strutturato, facilitandone la comprensione e la manutenzione.
- **Gestione Chiavi Flessibile:** Il server permette ai client di inviare la propria chiave pubblica e fornisce la propria su richiesta, un meccanismo base per l'instaurazione di comunicazioni sicure.

Punti Deboli e Aree di Miglioramento/Estensione:

- **Performance della Crittografia Asimmetrica:** Cifrare ogni messaggio (o frammento) con RSA è computazionalmente intensivo. Per applicazioni ad alto volume, un approccio ibrido (RSA per scambiare una chiave simmetrica, e AES per la cifratura dei dati successivi) sarebbe più performante.

- **Mancanza di Integrità e Autenticazione dei Messaggi (esplicita):** Sebbene OAEP offra una certa integrità, una firma digitale (es. RSA-PSS) o un HMAC (Hash-based Message Authentication Code) sul messaggio (o sui frammenti) prima della cifratura fornirebbe garanzie più forti sull'autenticità del mittente e sull'integrità del messaggio contro manomissioni attive. L'attuale autenticazione del client è implicita, basata sul possesso della chiave privata corrispondente alla pubblica inviata.
- **ACK e Affidabilità Limitati:** Il meccanismo di ACK conferma solo la ricezione e decifratura. Non esiste un sistema di NACK (Negative Acknowledgement) o di richiesta di ritrasmissione per frammenti persi. Per una maggiore affidabilità, sarebbe necessario implementare tali meccanismi.
- **Vulnerabilità a Replay Attack:** Non sono presenti numeri di sequenza o timestamp all'interno del payload cifrato per prevenire che un attaccante possa intercettare e reinviare messaggi validi.
- **Frammentazione in Uscita Non Implementata in `send_to`:** La variabile `self.max_fragment_size` è definita, ma la logica di frammentazione dei messaggi in uscita dal server non è presente nel metodo `send_to`. Attualmente, il server può solo inviare messaggi che rientrano nella dimensione massima del payload cifrabile con RSA e inviabile via UDP in un singolo pacchetto.
- **Gestione Stato Clienti:** Le chiavi pubbliche dei client sono memorizzate in un dizionario. Per un server a lunga esecuzione con molti client, potrebbe essere necessaria una gestione più sofisticata dello stato (es. persistenza, timeout per le chiavi inattive).

Valutazione Complessiva: `server_UDP_crypto_protocollo.py` è un eccellente esempio didattico che dimostra con successo l'integrazione di crittografia asimmetrica e gestione della frammentazione in un server UDP. Fornisce una base solida per la costruzione di applicazioni che richiedono comunicazioni UDP sicure. Sebbene presenti alcune limitazioni che ne sconsiglierebbero l'uso diretto in ambienti di produzione altamente critici o ad altissime prestazioni senza ulteriori modifiche, i concetti implementati sono corretti e ben presentati. Come strumento di apprendimento e prototipazione, il codice è di grande valore. Le aree di miglioramento suggerite potrebbero trasformarlo in una soluzione ancora più robusta e versatile.

7 File di Riferimento

- `server_UDP_crypto_protocollo.py`