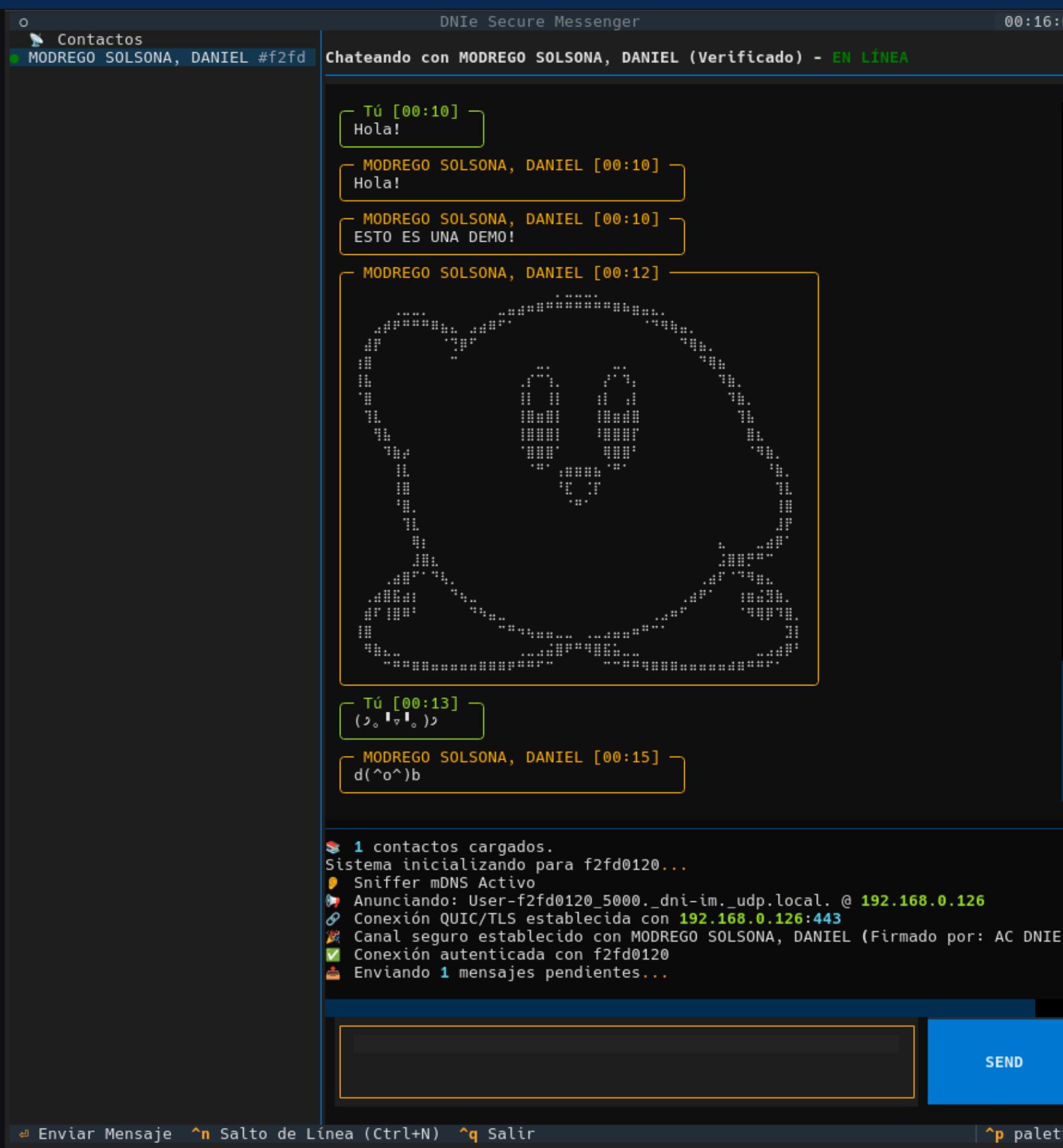


# SECURE INSTANT MESSAGING WITH DNI E IDENTITY

DANIEL MODREGO, ANA MONTUENGA  
CURSO 2025-2026

# OBJETIVOS

Crear un sistema de mensajería para usuarios en la red local, donde cada usuario tiene que identificarse con DNIe.



# FUNCIONALIDADES IMPLEMENTADAS

- Identidad DNIe:** Autenticación mutua validando la cadena de confianza de los certificados X.509 y firma digital en hardware.
- Protocolo Criptográfico:** Implementación de Noise IK (X25519 + ChaCha20-Poly1305) con Perfect Forward Secrecy (PFS).
- Transporte Seguro:** Comunicación P2P sobre QUIC (TLS 1.3) con protección Anti-Replay mediante ventana deslizante.
- Integridad y Fiabilidad:** Verificación con Poly1305, timestamps y confirmación de entrega (ACKs).
- Persistencia Privada:** Base de datos local cifrada completamente con Fernet (AES) usando claves derivadas de la tarjeta.

# FASES

<b>Fase 1: Arranque y Autenticación (Login con DNIe)</b>	<b>Fase 2: Inicialización del Sistema</b>	<b>Fase 3: Descubrimiento de Usuarios (Discovery)</b>	<b>Fase 4: Selección de Usuario y Handshake</b>	<b>Fase 5: Intercambio de Mensajes</b>	<b>Fase 6: Cierre seguro y Zeroización</b>
--	---	---	---	--	--

# FASE 1: ARRANQUE Y AUTENTICACIÓN (LOGIN CON DNIE)

## Generación de Clave Efímera

- Curva Elíptica X25519 (Alta velocidad).
- Perfect Forward Secrecy (PFS): Claves nuevas en cada ejecución.
- Security by Design: Key reside solo en RAM.

## Login en Tarjeta (PIN)

- Interfaz PKCS#11 (OpenSC).
- Autenticación de dos factores (Lo que tienes + Lo que sabes).
- Acceso exclusivo al token hardware.

## Firmas Criptográficas

- Firma RSA-SHA256 dentro del chip.
- Vinculación Criptográfica: Clave Efímera <-> DNIE
- Derivación de Clave de Cifrado.
- Prevención de suplantación.

```
key_priv = x25519.X25519PrivateKey.generate()  
  
# Obtenemos la clave pública para que el DNIE la firme  
key_pub_bytes = key_priv.public_key().public_bytes(  
    encoding=serialization.Encoding.Raw,  
    format=serialization.PublicFormat.Raw  
)
```

```
def authenticate(self, pin, max_retries=3):  
    """'Login' en la tarjeta usando el PIN del usuario."""  
    import time  
    for attempt in range(max_retries):  
        try:  
            # Abrir sesión autenticada  
            self.session = self.token.open(user_pin=pin)  
        return # Éxito, salir del bucle
```

# FASE 1: ARRANQUE Y AUTENTICACIÓN (LOGIN CON DNIE)

## Generación de Clave Efímera

- Curva Elíptica X25519 (Alta velocidad).
- Perfect Forward Secrecy (PFS): Claves nuevas en cada ejecución.
- Security by Design: Key reside solo en RAM.

## Login en Tarjeta (PIN)

- Interfaz PKCS#11 (OpenSC).
- Autenticación de dos factores (Lo que tienes + Lo que sabes).
- Acceso exclusivo al token hardware.

## Firmas Criptográficas

- Firma RSA-SHA256 dentro del chip.
- Vinculación Criptográfica: Clave Efímera <-> DNIE
- Derivación de Clave de Cifrado.
- Prevención de suplantación.

```
def sign_data(self, data_bytes):
    """
    Args: data_bytes (bytes): Datos a firmar.
    Returns: bytes: Firma digital RSA.
    """
    priv_keys = list(self.session.get_objects({
        Attribute.CLASS: ObjectClass.PRIVATE_KEY,
        Attribute.KEY_TYPE: pkcs11.KeyType.RSA
    }))
    for k in priv_keys:
        label = self._get_label(k)
        if "Autenticacion" in label or "Authentication" in label:
            target_key = k
            break
    # Firmar usando PKCS#1 v1.5 padding y SHA256
    signature = target_key.sign(
        data_bytes,
        mechanism=Mechanism.SHA256_RSA_PKCS
    )
    return bytes(signature)
```

# FASE 2: INICIALIZACIÓN DEL SISTEMA

## Derivación Clave Storage

- HKDF (HMAC-based Key Derivation Function).
- Conversión RSA (Asimétrico) -> AES (Simétrico).
- Determinismo: Mismo DNIe = Misma Clave DB.

## Inicio DB & Crypto Session

- Encryption at Rest (Fernet/AES-CBC).
- Inicio de tablas (Contacts, Messages).
- Preparación de mapas de memoria para sesiones.

## Inicio Servidor QUIC

- Generación de Certificados TLS Efímeros.
- Stream 0: Señalización y Handshake Noise.
- Streams N: Datos de chat cifrados.

```
def derive_storage_key(signature_bytes):
    """
    Deriva una clave simétrica robusta a partir de una firma digital.

    Utiliza HKDF (HMAC-based Key Derivation Function) para transformar la firma
    RSA de alta entropía generada por el DNIe en una clave de 32 bytes
    apta para cifrado simétrico AES/Fernet.
    """
    hkdf = HKDF(
        algorithm=hashes.SHA256(),
        length=32,
        salt=None, # No usamos salt porque la fuente (firma RSA) ya tiene alta entropía
        info=b'DNIe-Storage-Encryption-Key', # Contexto para la derivación
    )
    return bytearray(hkdf.derive(signature_bytes))

storage_key = derive_storage_key(storage_signature)

# Inicialización del almacenamiento cifrado
storage = Storage(key_bytes=storage_key, data_dir=args.data)
await storage.init()

# Gestor de sesiones criptográficas Noise
sessions = SessionManager(local_static_key, storage, local_proofs=proofs)
```

# FASE 2: INICIALIZACIÓN DEL SISTEMA

## Derivación Clave Storage

- HKDF (HMAC-based Key Derivation Function).
- Conversión RSA (Asimétrico) -> AES (Simétrico).
- Determinismo: Mismo DNIe = Misma Clave DB.

## Inicio DB & Crypto Session

- Encryption at Rest (Fernet/AES-CBC).
- Inicio de tablas (Contacts, Messages).
- Preparación de mapas de memoria para sesiones.

## Inicio Servidor QUIC

- Generación de Certificados TLS Efímeros.
- Stream 0: Señalización y Handshake Noise.
- Streams N: Datos de chat cifrados.

```
def generate_quic_tls_cert(data_dir):
    """
    Genera un certificado autofirmado y clave privada para el transporte QUIC.
    Returns: tuple: (ruta_certificado, ruta_clave_privada)
    """

    key = rsa.generate_private_key(public_exponent=65537, key_size=2048)
    subject = issuer = x509.Name([x509.NameAttribute(NameOID.COMMON_NAME, u"DNIe-Mesh-Node")])

    cert = (
        x509.CertificateBuilder()
        .subject_name(subject)
        .issuer_name(issuer)
        .public_key(key.public_key())
        .serial_number(x509.random_serial_number())
        .not_valid_before(datetime.datetime.now(datetime.timezone.utc))
        .not_valid_after(datetime.datetime.now(datetime.timezone.utc) + datetime.timedelta(days=1))
        .add_extension(
            x509.SubjectAlternativeName([
                x509.DNSName(u"localhost"),
                x509.IPAddress(ipaddress.IPv4Address("127.0.0.1")),
            ]),
            critical=False
        )
        .sign(key, hashes.SHA256())
    )
    cert_path = os.path.join(data_dir, "quic_cert.pem")
    key_path = os.path.join(data_dir, "quic_key.pem")
    return cert_path, key_path
```

# FASE 2: INICIALIZACIÓN DEL SISTEMA

## Derivación Clave Storage

- HKDF (HMAC-based Key Derivation Function).
- Conversión RSA (Asimétrico) -> AES (Simétrico).
- Determinismo: Mismo DNIe = Misma Clave DB.

## Inicio DB & Crypto Session

- Encryption at Rest (Fernet/AES-CBC).
- Inicio de tablas (Contacts, Messages).
- Preparación de mapas de memoria para sesiones.

## Inicio Servidor QUIC

- Generación de Certificados TLS Efímeros.
- Stream 0: Señalización y Handshake Noise.
- Streams N: Datos de chat cifrados.

```
# Inicialización del gestor de red QUIC
network_manager = QuicNetworkManager(
    session_manager=sessions,
    cert_path=cert_path,
    key_path=key_path,
    on_message=lambda a, m: None, # Se vinculará con la TUI después
    on_log=lambda t: print(f"LOG: {t}")
)

discovery = None

try:
    print(f"🔌 Iniciando servidor QUIC en {args.bind}:{args.port}...")
    await network_manager.start_server(args.bind, args.port)
    print(f"✅ Servidor QUIC activo en {args.bind}:{args.port}")

```

# FASE 3: DESCUBRIMIENTO DE USUARIOS (DISCOVERY)

## Anuncio Multicast

- Zeroconf / Bonjour / Avahi.
- Multicast DNS (UDP 5353).
- Publicación de Clave Pública pasiva.

## Sniffer de Paquetes

- Low-level Packet Inspection.
- Filtrado eficiente de bytes.
- Actualización de Routing Table en tiempo real.

```
# Iniciar Zeroconf (Anuncio)
interfaces = InterfaceChoice.All
if bind_ip and bind_ip != "0.0.0.0": interfaces = [bind_ip]
try:
    self.aiozc = AsyncZeroconf(interfaces=interfaces, ip_version=IPVersion.V4Only)
except:
    self.aiozc = AsyncZeroconf(interfaces=InterfaceChoice.All)

local_ip = bind_ip if (bind_ip and bind_ip != "0.0.0.0") else self.get_local_ip()
service_name = f"User-{self.unique_instance_id}_{self.port}.{MDNS_TYPE}"

desc = {'pub': self.pubkey_b64, 'user': clean_username}
self.info = ServiceInfo(
    MDNS_TYPE, service_name, addresses=[socket.inet_aton(local_ip)],
    port=self.port, properties=desc, server=f"{socket.gethostname()}.local."
)

self.on_log(f"📣 Anunciando: {service_name} @ {local_ip}")
await self.aiozc.async_register_service(self.info)

# Tarea periódica para detectar cambios de red
self._polling_task = asyncio.create_task(self._active_polling())
```

# FASE 3: DESCUBRIMIENTO DE USUARIOS (DISCOVERY)

## Anuncio Multicast

- Zeroconf / Bonjour / Avahi.
- Multicast DNS (UDP 5353).
- Publicación de Clave Pública pasiva.

## Sniffer de Paquetes

- Low-level Packet Inspection.
- Filtrado eficiente de bytes.
- Actualización de Routing Table en tiempo real.

```
class DiscoveryService:  
    async def start(self, username, bind_ip=None):  
        """Inicia el anuncio y la escucha."""  
        self.loop = asyncio.get_running_loop()  
        self.bind_ip = bind_ip  
        clean_username = username.replace("User-", "")  
        self.unique_instance_id = clean_username  
  
        # Iniciar Sniffer (Escucha)  
        try:  
            sock = socket.socket(socket.AF_INET, socket.SOCK_DGRAM, socket.IPPROTO_UDP)  
            sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)  
            try: sock.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEPORT, 1)  
            except: pass  
            sock.bind(('', 5353))  
  
            self.sniffer_transport, self.sniffer_protocol =  
                await self.loop.create_datagram_endpoint(  
                    lambda: RawSniffer(self), sock=sock  
                )  
            self.on_log("💡 Sniffer Activo")  
        except Exception as e:  
            self.on_log(f"⚠ Fallo al iniciar Sniffer: {e}")
```

# FASE 4: HANDSHAKE Y ESTABLECIMIENTO DE LA CONEXIÓN

## 1. Transporte Blindado (Capa Externa)

- Túnel QUIC TLS 1.3 establecido en 1-RTT (7 mensajes).
- Usa certificados efímeros para ocultar el protocolo interno.

## 2. Autenticación & Privacidad (Noise IK)

- Vínculo Legal: Firma RSA del chip DNIe valida la clave de sesión.

## 3. Cifrado de Chat (Simétrico)

- Negociación Diffie-Hellman + HKDF genera claves idénticas para Alice y Bob.
- Tráfico protegido con ChaCha20-Poly1305 (AEAD) + Túnel TLS 1.3.

Túnel QUIC (Cifrado TLS 1.3 con Certs Temp)

Protocolo Noise IK (Stream 0)  
-> Intercambio de Claves DNIe  
-> Verificación de Firmas

Mensajes de Chat (Streams 1, 2...)  
(Cifrados con ChaCha20 derivado de Noise)

# FASE 4: HANDSHAKE Y ESTABLECIMIENTO DE LA CONEXIÓN

## 1. Transporte Blindado (Capa Externa)

- Túnel QUIC TLS 1.3 establecido en 1-RTT (7 mensajes).
- Usa certificados efímeros para ocultar el protocolo interno.

## 2. Autenticación & Privacidad (Noise IK)

- Vínculo Legal: Firma RSA del chip DNIe valida la clave de sesión.

## 3. Cifrado de Chat (Simétrico)

- Negociación Diffie-Hellman + HKDF genera claves idénticas para Alice y Bob.
- Tráfico protegido con ChaCha20-Poly1305 (AEAD) + Túnel TLS 1.3.

## Establecimiento del Túnel QUIC TLS 1.3 (Capa de Transporte)

### 1. Pre-requisito: Credenciales Efímeras

- Generación automática de un certificado X.509 autofirmado y temporal (validez 24h) al iniciar la aplicación.
- Identidad genérica: CN=DNle-Mesh-Node (No identifica al usuario real).

### 2. Intercambio 1-RTT (Un solo viaje)

- Cliente → Servidor: Envía ClientHello + Key Share (Diffie-Hellman) + ALPN dnie-mesh-v1.
- Servidor → Cliente: Responde ServerHello + Key Share + Certificate (Efímero) + Finished.
- Cliente: Cierra con Finished.

### 3. Características de Seguridad

- Verificación Desactivada: El cliente acepta el certificado efímero sin validar CA (verify\_mode=False), delegando la autenticación real a la capa Noise interna.
- Cifrado Robusto: Todo el tráfico posterior viaja protegido por algoritmos AEAD estándar de TLS 1.3.

# FASE 4: HANDSHAKE Y ESTABLECIMIENTO DE LA CONEXIÓN

## 1. Transporte Blindado (Capa Externa)

- Túnel QUIC TLS 1.3 establecido en 1-RTT (7 mensajes).
- Usa certificados efímeros para ocultar el protocolo interno.

## 2. Autenticación & Privacidad (Noise IK)

- Vínculo Legal: Firma RSA del chip DNIe valida la clave de sesión.

## 3. Cifrado de Chat (Simétrico)

- Negociación Diffie-Hellman + HKDF genera claves idénticas para Alice y Bob.
- Tráfico protegido con ChaCha20-Poly1305 (AEAD) + Túnel TLS 1.3.

```
# Configuración TLS 1.3 Servidor
server_config = QuicConfiguration(
    is_client=False,
    alpn_protocols=["dnie-mesh-v1"] # Protocolo Aplicación
)
# Carga certificado efímero autofirmado (generado al inicio)
server_config.load_cert_chain("data/quic_cert.pem", "data/quic_key.pem")

# Iniciar escucha UDP
await serve(
    host="0.0.0.0",
    port=443,
    configuration=server_config,
    create_protocol=MeshQuicProtocol
)

# Configuración TLS 1.3 Cliente
client_config = QuicConfiguration(
    is_client=True,
    alpn_protocols=["dnie-mesh-v1"]
)
# IMPORTANTE: No verificar CA (certificado es efímero/autofirmado)
client_config.verify_mode = False

# Establecer Túnel (Handshake 1-RTT)
async with connect(
    ip,
    port,
    configuration=self._client_config,
    create_protocol=protocol_factory,
) as protocol:
    # Al entrar aquí, el túnel TLS 1.3 ya es seguro.
    # Ahora inicia el handshake interno Noise IK...
    pass
```

# FASE 4: HANDSHAKE Y ESTABLECIMIENTO DE LA CONEXIÓN

## 1. Transporte Blindado (Capa Externa)

- Túnel QUIC TLS 1.3 establecido en 1-RTT (7 mensajes).
- Usa certificados efímeros para ocultar el protocolo interno.

## 2. Autenticación & Privacidad (Noise IK)

- Vínculo Legal: Firma RSA del chip DNIe valida la clave de sesión.

## 3. Cifrado de Chat (Simétrico)

- Negociación Diffie-Hellman + HKDF genera claves idénticas para Alice y Bob.
- Tráfico protegido con ChaCha20-Poly1305 (AEAD) + Túnel TLS 1.3.

```
def _dh(self, priv, pub):
    """Realiza intercambio Diffie-Hellman (X25519) raw."""
    return priv.exchange(pub)

def _kdf(self, km, material):
    """
    Función de Derivación de Claves (HKDF).
    Usa BLAKE2s para derivar nuevas claves de encadenamiento y cifrado.
    Retorna bytarray para poder zeroizar después.
    """
    hkdf = HKDF(algorithm=hashes.BLAKE2s(digest_size=32), length=64, salt=bytes(km), info=b'')
    output = bytearray(hkdf.derive(material))
    return output[:32], output[32:] # Retorna (nueva_ck, clave_cifrado) como bytearray

def _mix_key(self, ck, dh_out):
    """Mezcla el resultado de un DH en el estado (Stateful Hash)."""
    return self._kdf(ck, dh_out)

def initialize(self):
    """Genera claves efímeras y finaliza el hash inicial."""
    self.e_priv = x25519.X25519PrivateKey.generate()
    self.e_pub = self.e_priv.public_key()
    # Almacenamos ck como bytearray para poder zeroizarlo después
    self.ck = bytearray(self.chaining_key.finalize())

def _prepare_identity_payload(self):
    """Serializa las pruebas (Certificado + Firma) a JSON bytes."""
    if not self.local_proofs:
        return b'{}'
    return json.dumps(self.local_proofs).encode('utf-8')
```

# DIAGRAMA NOISE IK (HANDSHAKE)

**ALICE**

**Claves Efímeras:**  $e\_priv\_A, e\_pub\_A$

**Claves Estáticas:**  $s\_priv\_A, \underbrace{s\_pub\_A}_{mDNS}$

**BOB**

**Claves Efímeras:**  $e\_priv\_B, e\_pub\_B$

**Claves Estáticas:**  $s\_priv\_B, \underbrace{s\_pub\_B}_{mDNS}$

$$CK_0 = \text{Hash}(\text{"DNIe-IM-V2-Signed"})$$

$$1^{\circ} \left\{ \begin{array}{l} CK_1, K_S = \mathbf{KDF}(CK_0, \mathbf{DH}(e\_priv\_A, s\_pub\_B)) \\ CK_2, K_{PA} = \mathbf{KDF}(CK_1, \mathbf{DH}(s\_priv\_A, s\_pub\_B)) \end{array} \right\} 2^{\circ} \left\{ \begin{array}{l} CK_1, K_S = \mathbf{KDF}(CK_0, \mathbf{DH}(s\_priv\_B, e\_pub\_A)) \\ CK_2, K_{PA} = \mathbf{KDF}(CK_1, \mathbf{DH}(s\_priv\_B, s\_pub\_A)) \end{array} \right\}$$

$$4^{\circ} \left\{ \begin{array}{l} CK_3, \sim = \mathbf{KDF}(CK_2, \mathbf{DH}(e\_priv\_A, e\_pub\_B)) \\ CK_4, K_{PB} = \mathbf{KDF}(CK_3, \mathbf{DH}(s\_priv\_A, e\_pub\_B)) \end{array} \right\} 3^{\circ} \left\{ \begin{array}{l} CK_3, \sim = \mathbf{KDF}(CK_2, \mathbf{DH}(e\_priv\_B, e\_pub\_A)) \\ CK_4, K_{PB} = \mathbf{KDF}(CK_3, \mathbf{DH}(e\_priv\_B, s\_pub\_A)) \end{array} \right\}$$

$$K_{B \rightarrow A}, K_{A \rightarrow B} = \mathbf{KDF}(CK_4)$$

**ALICE INICIA HANDSHAKE**

0x01 1B	Size 4B	IdxA 4B	$e\_pub\_A$ 32B	$s\_pub\_A$ 32B	Tag 16B	<b>JSON: Cert X.509 + Firma de e-pub-A</b>	Tag 16B
------------	------------	------------	--------------------	--------------------	------------	--	------------

ChaCha20Poly1305( $K_S$ )      ChaCha20Poly1305( $K_{PA}$ )

**BOB RESPONDE HANDSHAKE**

0x02 1B	Size 4B	IdxB 4B	IdxA 4B	$e\_pub\_B$ 32B	<b>JSON: Cert X.509 + Firma de e-pub-B</b>	Tag 16B
------------	------------	------------	------------	--------------------	--	------------

ChaCha20Poly1305( $K_{PB}$ )

# FASE 4: HANDSHAKE Y ESTABLECIMIENTO DE LA CONEXIÓN

## 1. Transporte Blindado (Capa Externa)

- Túnel QUIC TLS 1.3 establecido en 1-RTT (7 mensajes).
- Usa certificados efímeros para ocultar el protocolo interno.

## 2. Autenticación & Privacidad (Noise IK)

- Vínculo Legal: Firma RSA del chip DNIe valida la clave de sesión.

## 3. Cifrado de Chat (Simétrico)

- Negociación Diffie-Hellman + HKDF genera claves idénticas para Alice y Bob.
- Tráfico protegido con ChaCha20-Poly1305 (AEAD) + Túnel TLS 1.3.

```
def verify_peer_identity(x25519_pub_key, proofs):
    """Verifica criptográficamente la identidad de un par remoto."""
    try:
        cert_bytes = bytes.fromhex(proofs['cert'])
        signature_bytes = bytes.fromhex(proofs['sig'])
        peer_cert = x509.load_der_x509_certificate(cert_bytes)
        rsa_pub_key = peer_cert.public_key()

        # 1. Verificación Temporal
        now = datetime.now(timezone.utc)
        if now > peer_cert.not_valid_after_utc:
            raise Exception("El certificado ha CADUCADO")

        # 2. Verificación de Uso de Clave (Key Usage)
        try:
            key_usage_ext = peer_cert.extensions.get_extension_for_class(x509.KeyUsage)
            usage = key_usage_ext.value

            # 3. Verificación de Cadena de Confianza (Chain of Trust)
            issuer_name = "CA Desconocida (Sin Verificación)"
            if not GLOBAL_TRUST_STORE:
                issuer_name = "NO-CONFIAL/ SIN-ALMACEN"
                # Permitimos continuar sin validación CA solo si no hay CAs cargadas (Modo inseguro)
                is_trusted = True
            else:
                for ca_cert in GLOBAL_TRUST_STORE:
                    try:
                        ca_public_key = ca_cert.public_key()
                        # Verificamos la firma criptográfica del certificado del par con la clave de la CA
                        ca_public_key.verify(peer_cert.signature, peer_cert.tbs_certificate_bytes,
                                             padding.PKCS1v15(), peer_cert.signature_hash_algorithm)
                        is_trusted = True
                        issuer_name = get_common_name(ca_cert)
                        print(f"🔒 [SEGURIDAD] Certificado DNIe verificado correctamente por: {issuer_name}")
                        break
                    except:
                        continue
        except:
            pass

        # 4. Verificación de la Firma sobre la Clave Efímera
        data_to_verify = x25519_pub_key.public_bytes(encoding=serialization.Encoding.Raw,
                                                       format=serialization.PublicFormat.Raw)
        rsa_pub_key.verify(signature_bytes, data_to_verify, padding.PKCS1v15(), hashes.SHA256())
        real_name = get_common_name(peer_cert).replace("(AUTENTICACIÓN)", "").strip()
        return real_name, issuer_name
    except:
        pass
```

# FASE 5: INTERCAMBIO DE MENSAJES

## Envío Cifrado (Type 0x03)

- **JSON Serialization.**
- **Authenticated Encryption (AEAD).**
- **Packet Type header (0x03).**

## Recepción y ACK

- **Anti-Replay Window (Sliding bitmap).**
- **Reliability Layer sobre UDP (ACKs).**
- **Integridad de Hash.**

```
def send_chat_message(self, msg_struct: dict) -> bool:  
    """  
    Envía un mensaje de chat.  
    """  
  
    try:  
        # 1. Serializar el JSON a bytes  
        json_bytes = json.dumps(msg_struct).encode('utf-8')  
  
        # 2. Obtener y preparar el Nonce (12 bytes: 8 bytes contador + 4 bytes padding ceros)  
        nonce = struct.pack('<Q', self.noise_session.tx_nonce) + b'\x00\x00\x00\x00'  
  
        # 3. Cifrar usando el encryptor de la sesión  
        ciphertext = self.noise_session.encryptor.encrypt(nonce, json_bytes, None)  
  
        # 4. Incrementar el contador local para el siguiente mensaje  
        self.noise_session.tx_nonce += 1  
  
        # 5. Empaquetar para envío: Enviamos [Nonce (12B)] + [Ciphertext]  
        # Es vital enviar el nonce para que el receptor sepa cuál usar  
        final_payload = nonce + ciphertext  
  
        # 6. Enviar por el stream de QUIC  
        stream_id = self._quic.get_next_available_stream_id()  
        self._quic.send_stream_data(stream_id, final_payload, end_stream=True)  
        self.transmit()  
  
        # Guardar mensaje como pendiente de ACK (usando msg_struct original)  
        if 'id' in msg_struct:  
            self._unacked_messages[msg_struct['id']] = msg_struct.copy()  
  
    return True  
except Exception as e:  
    ...  
    return False
```

# FASE 5: INTERCAMBIO DE MENSAJES

## Envío Cifrado (Type 0x03)

- JSON Serialization.
- Authenticated Encryption (AEAD).
- Packet Type header (0x03).

## Recepción y ACK

- Anti-Replay Window (Sliding bitmap).
- Reliability Layer sobre UDP (ACKs).
- Integridad de Hash.

```
def _send_msg_ack(self, msg_id: str):
    """Envía ACK de recepción de mensaje."""
    self._send_signal(SIGNAL_MSG_ACK, msg_id.encode('utf-8'))

def _handle_chat_data(self, stream_id: int, data: bytes, end_stream: bool):
    """Procesa datos de chat recibidos"""
    ...
    # Stream completo - procesar datos acumulados
    full_data = self._chat_buffers.pop(stream_id, b"")
    ...
    # Verificar si el contador es válido usando ventana deslizante
    if not self._check_replay(counter):
        self._log(f"⚠️ Paquete rechazado por anti-replay (contador: {counter})")
        return
    ...
    # 4. Actualizar ventana anti-replay DESPUÉS de descifrado exitoso
    self._update_replay_window(counter)
    ...
    # Enviar ACK al emisor
    if 'id' in msg_struct:
        self._send_msg_ack(msg_struct['id'])

def _handle_msg_ack(self, payload: bytes):
    """Procesa ACK de mensaje recibido del peer."""
    try:
        msg_id = payload.decode('utf-8')

        # Remover de mensajes sin ACK
        if msg_id in self._unacked_messages:
            del self._unacked_messages[msg_id]

        # Notificar a la capa superior
        if self.network_manager and self.network_manager.on_ack_received:
            # Obtener dirección correcta del peer
            peer_addr = None
            if self.peer_identity:
                peer_addr = self.network_manager.peer_addresses.get(self.peer_identity)
            if not peer_addr:
                ...
            self.network_manager.on_ack_received(peer_addr, msg_id)
    except Exception as e:
        pass # ACK fallido, el mensaje se reenviará si es necesario
```

# FASE 6: GUARDADO Y SALIDA

## Zeroization & Shutdown

- **Memory Scrubbing (Zeroization).**
- **Graceful Shutdown.**
- **Notificación de desconexión a pares.**

```
finally:  
    # --- Bloque de Cierre Limpio ---  
    print("\n🔴 Cerrando aplicación y liberando recursos...")  
  
    # 1. Borrar de forma segura todas las sesiones criptográficas  
    if sessions:  
        sessions.zeroize_all_sessions()  
        print("🔒 Sesiones criptográficas borradas de forma segura.")  
  
    # 2. Cerrar conexión a base de datos (incluye borrado de clave)  
    await storage.close()  
  
    if network_manager:  
        # 3. Enviar mensaje de "Desconexión" cifrado a los pares  
        # activos y cerrar conexiones QUIC  
        await network_manager.broadcast_disconnect()  
        await network_manager.close()  
  
    if discovery:  
        # 4. Detener anuncios mDNS y salir del grupo multicast  
        await discovery.stop()  
  
    # 5. Borrar la clave de almacenamiento derivada (ya es bytearray)  
    if storage_key:  
        try:  
            zeroize1(storage_key)  
            print("🔒 Clave de almacenamiento derivada borrada.")  
        except Exception:  
            pass  
  
    print("👋 Bye! (Ejecución finalizada)")
```

# FASE 6: GUARDADO Y SALIDA

## Zeroization & Shutdown

- **Memory Scrubbing (Zeroization).**
- **Graceful Shutdown.**
- **Notificación de desconexión a pares.**

```
def zeroize_session(self):
    """
        Limpia las claves criptográficas de la sesión de forma segura.

        Debe llamarse cuando:
        - La sesión se cierra explícitamente
        - El peer se desconecta
        - La aplicación se cierra

        NOTA: Las claves X25519 de cryptography no pueden ser zeroizadas
        porque la librería no expone los bytes internos de forma mutable.
        Solo podemos eliminar las referencias y confiar en el GC.

        El chaining key (ck) SÍ se almacena como bytearray y puede ser zeroizado.

        # Borrar chaining key (almacenada como bytearray)
        if hasattr(self, 'ck') and self.ck is not None:
            if isinstance(self.ck, bytearray):
                zeroize1(self.ck)
            self.ck = None

        # Eliminar referencias a claves (no podemos zeroizar los bytes internos)
        self.e_priv = None
        self.encryptor = None
        self.decryptor = None
        self.rs_pub = None
        self.re_pub = None

        # Resetear contadores
        self.tx_nonce = 0
        self.rx_nonce = 0
        self.replay_bitmap = 0
```

MUCHAS GRACIAS

```
    render() {
      return (
        <React.Fragment>
          <div className="py-5">
            <div className="container">
              <Title name="our" title="product">
              <div className="row">
                <ProductConsumer>
                  {(value) => {
                    |   |   |   console.log(value)
                  }}
                </ProductConsumer>
              </div>
            </div>
          </div>
        <React.Fragment>
```