Mi VPN

Emiliano A. Billi

Ver 1.0

Y me desperté con ganas de armar una VPN.
—¿Por qué?
—Y... ¿por qué no?

Introducción

No había una razón concreta, al menos no una que pudiera explicar fácilmente. Tengo un deseo constante de entender cómo funcionan las cosas, quizá es más sencillo decirlo de esta manera que tratar de inventar una necesidad o un propósito para hacerlo. Debo confesar que muchas veces fui juzgado de reinventar la rueda, ante esta mirada tengo que decir que, por suerte para mí, programar no es un trabajo.

Crear software es mi forma de expresarme, no programo porque deba, lo hago porque quiero, porque veo la posibilidad de tomar algo que solo está en mi cabeza y darle forma en el mundo real, aunque realmente sea virtual. Para mí, la verdadera satisfacción está en transformar la teoría, esa que a veces parece inalcanzable, en algo tangible. Y eso conlleva el desafío de saber que, aunque parezca complicado o improbable, se puede hacer. Es un juego constante entre lo que se puede imaginar y lo que se puede construir.

Otro rasgo de mi personalidad es que detesto los proyectos que utilizan librerías de terceros, prefiero siempre utilizar recursos estándares o que forman parte de un proyecto colaborativo. Crear desde cero, sin depender de soluciones externas, me obliga a entender cada capa, cada paso del proceso. Es una búsqueda de control, de poder decir que cada línea de código que escribo tiene un propósito definido por mí, y no por alguien más.

Un dato adicional es que siempre me han gustado los **protocolos**. Hay algo profundamente fascinante en ellos, algo que va más allá del simple intercambio de datos entre dos puntos. Es la idea de que dos programas, separados por distancias a veces insalvables, pueden entenderse, pueden trabajar juntos con un propósito en común. Ese nivel de coordinación, esa capacidad de interactuar, siempre me ha parecido algo mágico.

Un lenguaje, un acuerdo tácito entre ambos extremos, que les permite comunicarse en perfecta armonía. Esa magia de los protocolos es lo que me atrajo a este proyecto. Construir una VPN desde cero no es solo escribir código, es ser parte de ese entendimiento a la distancia, donde todo tiene que fluir de manera precisa para que funcione. Me motiva el desafío de que cada bit de información tenga su razón de ser, que nada sea dejado al azar.

Tenía muchos puntos que resolver: *extraer un paquete de la pila del sistema operativo*, *encapsular, transportar, autenticar, encriptar y, lo más complicado de todo, hacer que funcione,* esto último iba a ser lo más complicado. Había algo que jugaba a mi favor: el stack **TCP/IP** lo conozco de memoria. Esa familiaridad con los protocolos no solo me daba confianza, sino también la seguridad de que los problemas que fueran surgiendo tendrían solución.

Parte 1. El protocolo de transporte

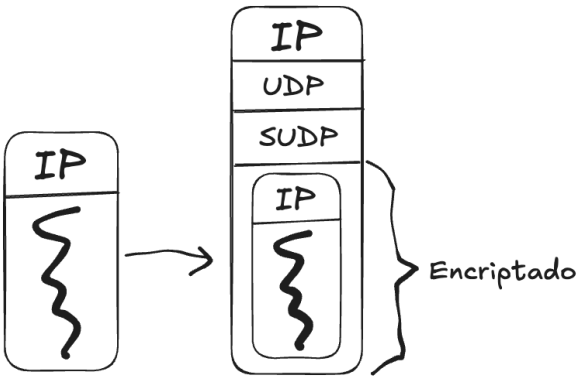
Empecemos con un poco de teoría. El propósito de un protocolo de transporte es asegurar que los paquetes IP lleguen de un extremo a otro de manera segura. Esto implica no solo transportar los datos, sino también autenticar y encriptar la comunicación. Podría haber optado por una solución simple: encapsular un paquete IP dentro de otro paquete IP. Sin embargo, esta idea se desmoronó rápidamente al enfrentar una realidad técnica: muchos enrutadores que operan entre el emisor y el receptor no permiten el paso de paquetes IP que no cumplan con un protocolo válido o definido por los estándares. Así que esa no era la opción.

La alternativa más sencilla hubiera sido encapsular el paquete IP sobre **TCP**, el protocolo de transporte confiable por excelencia. TCP maneja la retransmisión de paquetes perdidos, el control de flujo y el orden correcto de los datos. Sin embargo, esta solución también tenía sus inconvenientes. **TCP**, aunque robusto, añade bastante **overhead** debido a sus mecanismos de control y corrección. Además, usar TCP sobre TCP (en una VPN, los datos pueden viajar sobre TCP y luego volver a encapsularse en otro TCP) puede crear problemas de rendimiento conocidos como **TCP meltdown**, donde las capas de retransmisión de ambos TCP pueden entrar en conflicto, haciendo el tráfico más lento y menos eficiente.

Por estas razones, decidí armar mi propio protocolo de transporte a nivel de la capa de usuario, pero sobre **UDP**. Una de las grandes ventajas de UDP es que tiene muy poco overhead comparado con TCP. Al no preocuparse por retransmitir paquetes perdidos ni mantener el orden de los mismos, **UDP** es ideal para aplicaciones donde la velocidad y la latencia mínima son críticas, como en transmisiones en tiempo real. Además, como quería tener control total sobre el diseño y comportamiento de mi protocolo, UDP me ofrecía una mayor flexibilidad al no imponer las restricciones de TCP.

Hay, por supuesto, muchos protocolos seguros construidos sobre UDP, como **DTLS** o **QUIC**, pero la mayoría de ellos están diseñados para conexiones, y yo quería evitar justamente eso. Mi objetivo era construir un protocolo sin estado (hasta ahí nomas), sin la necesidad de establecer y mantener una conexión permanente. Esto no sólo reducía la complejidad, sino que además me permitía diseñar algo a mi manera, desde los cimientos. La idea de tener control total sobre el flujo de paquetes, la autenticación y el cifrado me resultaba irresistible. Aquí no se trataba sólo de implementar algo funcional, sino de crear un protocolo que se ajustara a mis propias reglas.

Listo, el transporte iba a ser **UDP**, pero no podía limitarme a usar **UDP raw**. Sabía que necesitaba agregar cierta información extra para asegurar el enlace, es decir, debía garantizar la seguridad, autenticación y confidencialidad de extremo a extremo. Así que la idea era clara: cada paquete IP debía ser encapsulado en un paquete UDP, pero con mi propio protocolo. Lo llamé **SUDP**



(Secure UDP). No fue el nombre más original, lo admito, pero reflejaba exactamente lo que buscaba: un protocolo que añadiera una capa de seguridad al transporte sin sacrificar la simplicidad y eficiencia de **UDP**.

El apretón de manos (Handshake)

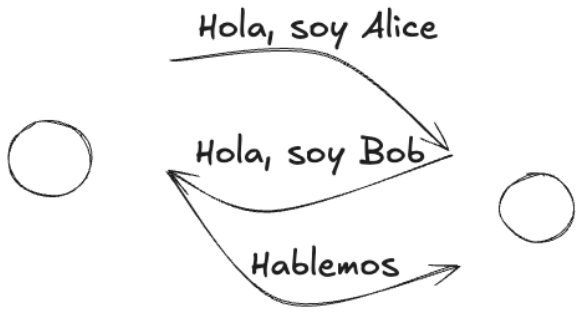
Para iniciar una comunicación de extremo a extremo, es fundamental garantizar la autenticidad de los interlocutores y verificar que ambos están disponibles. Si los interlocutores fueran humanos, las preguntas podrían sonar algo así:

—Hola, tengo ganas de iniciar una conversación. Soy Alice y aquí está mi identificación.
—Hola Alice, qué bueno verte por aquí, yo soy Bob y esta es mi identificación.
—Perfecto. Hablemos.

Este intercambio es un **handshake** de tres vías, muy similar al que realiza **TCP** para establecer una conexión. Pero, ¿por qué debe ser un **handshake** de tres vías y qué tipo de información debe intercambiarse?

Primero, Bob necesita asegurarse de que la persona que inició la conversación es realmente Alice, y viceversa. En los mensajes iniciales, ambos intercambian la información necesaria para acordar cómo encriptar los datos durante la comunicación. El tercer mensaje es clave para despejar cualquier duda. La pregunta que surge es: **¿Cómo sabe Bob que quien le está escribiendo es realmente Alice? ¿Cómo sabe Alice que quien respondió es Bob?**

La respuesta es: **autenticación**. En este protocolo, utilizamos **firmas digitales** basadas en el esquema de clave pública y privada. Alice y Bob ya conocen sus claves públicas respectivas, sin necesidad de intercambiarlas a través del enlace. Cuando Alice envía el saludo inicial, lo firma digitalmente. Bob, al recibirlo, verifica la firma y, sin lugar a dudas, sabe que el mensaje proviene de Alice. Luego, Bob firma su respuesta, la envía y así Alice también puede estar segura de que la respuesta proviene de Bob.



El primer mensaje del cliente es solo un saludo: *"Hola, soy Alice"*. El servidor responde confirmando que recibió el mensaje y que está listo: *"Hola, soy Bob"*. Pero aquí ocurre un detalle importante: **el servidor no sabe si su mensaje ha llegado al cliente**. No tiene certeza de que el cliente haya recibido su confirmación. Si la comunicación terminara aquí, el servidor estaría a ciegas, confiando en que todo salió bien, pero sin ninguna confirmación real.

Aquí es donde entra la importancia de la **tercera vía**: el cliente debe enviar un último mensaje diciendo: *"Hablemos",* y ahora ambos saben que están listos. Con ese último paso, el servidor tiene la certeza de que ambos lados están sincronizados y que la comunicación puede comenzar de manera confiable.

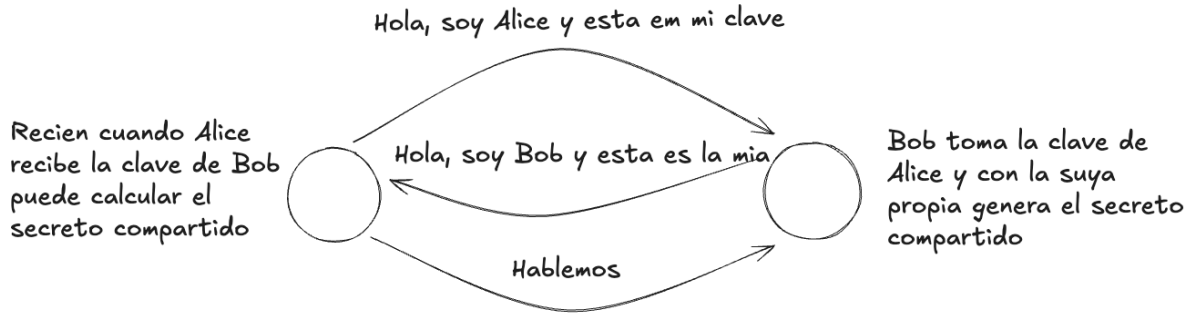
Este **último mensaje es clave** porque garantiza que el servidor no se quede esperando sin saber si el cliente está listo. Si se omitiera, el servidor podría estar enviando información al vacío. Esa tercera vía completa el círculo y asegura que **ambos lados saben que el otro está preparado y disponible para intercambiar datos**.

Sin esa última confirmación, el servidor estaría en una situación de incertidumbre. Por eso, el handshake de tres vías es más que una simple formalidad: **es un mecanismo que asegura que la comunicación es confiable y que ambos lados están listos para seguir adelante**.

El handshake de tres vías no es solo un saludo formal. Más allá de la sincronización inicial, es el momento donde ocurre algo crucial: el **intercambio de información que hará segura toda la comunicación posterior**. En este punto, **ambos extremos** (el cliente y el servidor) intercambian algo muy importante: sus **claves públicas efímeras**.

Este intercambio utiliza un sistema llamado **Diffie-Hellman**, que es un protocolo criptográfico diseñado para permitir que dos partes generen una **clave secreta compartida**, incluso si están comunicándose a través de un canal inseguro. Es como si ambos tuvieran una caja fuerte, cada uno con una clave única, pero en lugar de enviar las claves directamente, intercambian piezas de información que, combinadas con sus propias claves privadas, les permiten crear una clave secreta común. Ningún atacante que esté escuchando el intercambio puede descubrir esta clave secreta.

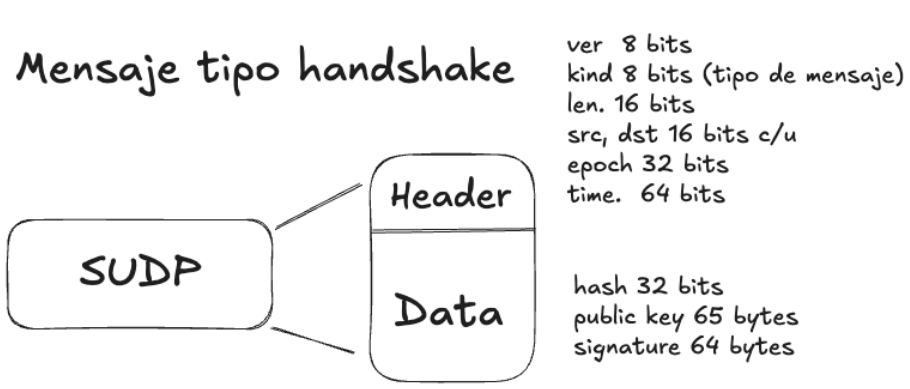
.



¿Por qué es esto importante? Porque esta **clave secreta** derivada es la que se utilizará para **encriptar** los paquetes IP que van a viajar entre ambos lados. Es decir, una vez que se completa el handshake, todo el tráfico de red entre el cliente y el servidor estará cifrado y protegido. De esta forma, aunque alguien pueda interceptar los paquetes en tránsito, no podrá leer el contenido sin esa clave secreta que solo conocen el cliente y el servidor.

Este intercambio de **claves efímeras** garantiza que incluso si alguien captura estas claves, no podrá utilizarlas para descifrar el tráfico en el futuro, ya que son **efímeras**: se utilizan solo durante esa sesión y después se desechan.

Así, el **handshake de tres vías** no solo asegura que ambos extremos están listos, sino que también establece la base de la **seguridad criptográfica**. Una vez completado, ambos lados comparten una **clave secreta** y pueden comenzar a cifrar los datos de manera segura. Esto es fundamental para que la VPN proteja la privacidad y la integridad de la comunicación.



La estructura del mensaje de handshake está diseñada específicamente para soportar este tipo de intercambio de información. El **header** es consistente y se mantiene fijo a lo largo del protocolo, siendo el campo **kind** el que indica qué tipo de información contiene el mensaje. Me gustaría destacar dos elementos clave: uno en la cabecera y otro en el **payload** (o área de datos); el **time** y el **hash**.

El **time** es simplemente el timestamp que indica el milisegundo en que se envió el mensaje. Una de las características importantes del protocolo es que no tolera mensajes antiguos; por lo tanto, los relojes entre los dos extremos deben estar sincronizados con una tolerancia de no más de 5 segundos. Puede parecer mucho tiempo, pero en realidad es una eternidad para este tipo de intercambios. Este valor es arbitrario y puede ajustarse fácilmente según sea necesario.

El segundo dato relevante es el **hash**, que se encuentra en el cuerpo del mensaje de handshake. Este hash es el resumen de la cabecera y se incluye al inicio del handshake para garantizar que tanto el header como los datos formen parte del mismo mensaje. La firma de este hash, junto con la clave pública de intercambio, asegura la integridad de ambos.

$$signature = (hash(cabecera) + Publickey)$$

Sin embargo, hay un detalle importante que tuve en cuenta: **podría ocurrir una colisión**. Es decir, es posible que en algún momento en el futuro se genere un timestamp que produzca el mismo hash. Esto es un problema, y soy consciente de ello. Lo mejoraré en el futuro, pero en realidad es algo más complicado de lo que parece: para que ocurra una colisión significativa, tendrían que existir **dos mensajes en el futuro que generen el mismo hash**, es decir, el primer y el tercer mensaje del handshake, con menos de 5 segundos de diferencia entre ellos. ¿Es posible? Sí, pero eso será un problema que como dije antes, para el Emiliano del futuro.

El mensaje de Control

Otro tipo de mensaje crucial es el **mensaje de control**, que puede tener diversas características según su propósito. Uno de los más importantes es el **keepalive**. Este mensaje se envía periódicamente desde el cliente al servidor por dos razones fundamentales: la primera es **mantener activa una conexión UDP** cuando el cliente está detrás de un **NAT**. Los dispositivos NAT tienden a cerrar conexiones inactivas después de un tiempo, lo que impide que los paquetes entrantes lleguen al cliente. El **keepalive** evita este problema al enviar mensajes pequeños que "mantienen viva" la conexión, asegurando que el NAT no cierre el puerto que se utiliza para recibir los paquetes entrantes.

La segunda razón para el mensaje de keepalive es dar señales de vida al servidor, indicando que el cliente sigue conectado, lo que evita que el servidor cierre la sesión por inactividad. Sin este mecanismo, el servidor podría asumir que el cliente se ha desconectado y finalizar la conexión.

Otro mensaje de control relevante es la **confirmación de epoch**, que es el tercer mensaje del handshake. Aquí es donde debemos aclarar que el **handshake** no ocurre solo al inicio de la comunicación. El cliente envía un nuevo handshake cada cierto intervalo de tiempo, siempre incrementando la **epoch**. Cada epoch tiene su propio secreto compartido, lo que refuerza la seguridad de la comunicación. Este proceso continuo asegura que la conexión no solo se mantenga viva, sino también segura a lo largo del tiempo, con nuevas claves criptográficas generadas regularmente.

El servidor por su lado registra el timestamp de la última vez que recibió un mensaje del cliente, al pasar determinado tiempo sin datos, este borra todo registro de la conexión.

Otra cosa que vale la pena mencionar es que ambos extremos se comportan de manera distinta. Por un lado, está el **cliente** en **modo activo** y **orientado a conexión**: es el que inicia la comunicación. Por el otro lado está el **servidor**, en **modo pasivo** y **orientado a mensajes**, que espera las solicitudes de los clientes. El servidor puede recibir mensajes de múltiples clientes simultáneamente, mientras que un cliente solo puede recibir mensajes del servidor. Unos campos de la cabecera especifican cual es el origen y destino del mismo (src y dst)

Pese a que esta diferencia entre el comportamiento de cliente y servidor es un aspecto fundamental de la especificación del protocolo se detalla más adelante en la sección de arquitectura.

Los datos

El **mensaje de datos** es uno de los más importantes dentro del protocolo, ya que transporta la información real que se desea enviar de manera segura. Para garantizar la **confidencialidad** de los datos, se utiliza el algoritmo de encriptación **AES-GCM** (Galois/Counter Mode), un modo de operación ampliamente reconocido por su eficiencia y seguridad en comunicaciones criptográficas.

Antes de que el mensaje sea encriptado, se incorpora el **hash del encabezado**. Este paso es fundamental porque asegura que tanto la cabecera como los datos están vinculados de forma segura. El hash del encabezado se genera sobre toda la cabecera del mensaje, lo que garantiza su integridad. Al incluir este hash dentro de los datos antes de la encriptación, nos aseguramos de que cualquier modificación en el encabezado sea detectada durante la verificación, ya que el hash no coincidirá.

encripted = (hash(cabecera) + texto plano)

De esta manera, el protocolo no solo garantiza que los datos se envíen de forma encriptada y segura, sino también que cualquier intento de manipulación o alteración de la cabecera del mensaje sea fácilmente detectable.

La implementación

Resolver el handshake y el inicio de la comunicación era lo más importante que tenía por delante. Ya había trazado cómo se comportaría el protocolo con los mensajes de control, y una vez que tuviera el secreto compartido, encriptar la información sería relativamente sencillo. Ahora me tocaba definir en qué lenguaje iba a programarlo.

El lenguaje de programación suele ser lo último que se define en muchos proyectos, pero aquí me encuentro en una contradicción. Aunque es evidente que primero se debe definir el protocolo y cómo va a interactuar, considero que es fundamental elegir el lenguaje **antes** de abordar otros aspectos. La razón es simple: las características del lenguaje influyen directamente en el diseño de la arquitectura del sistema. Por ejemplo, si decido implementarlo en **Python**, la arquitectura deberá ser completamente diferente a si lo hago en **Golang**, ya que cada lenguaje tiene sus propias fortalezas y limitaciones en cuanto a concurrencia, rendimiento y manejo de recursos.

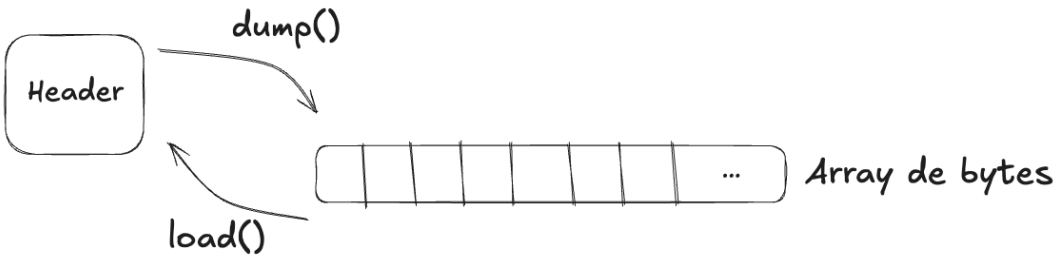
Mi primera opción fue **C**, un lenguaje que domino desde hace más de 20 años. Sin embargo, todo tiene sus límites. ¿Estaba dispuesto a implementar el intercambio de claves públicas para calcular un secreto compartido y usar las librerías necesarias para firmas y curvas ECDSA? La respuesta era sí. Pero aprender a utilizar esas librerías implicaba un esfuerzo adicional y, quizá, incluso sería digno de una precuela de este relato.

Entonces consideré mi segunda opción: **Golang**. Un lenguaje moderno, elegante, y sorprendentemente rápido. Además, maneja muy bien la concurrencia, es compilado y bastante portable. Tiene una librería estándar muy completa e interesante. Lo único que no me convencía del todo era su **runtime**, y la interacción con **C** (porque estaba seguro de que alguna parte tendría que escribirla en C) no es la más amigable.

Como decía mi abuelo: *"Nunca uses un thread en C sobre una goroutina de Golang si no entiendes cómo funciona el runtime"*. (Obviamente mi abuelo jamás dijo eso. No tenía ni idea de lo que era Golang, ni C, y dudo que haya conocido una computadora).

Veamos cómo luce el código de la cabecera y sus dos funciones principales: `load` y `dump`. Es crucial entender que, para que un paquete pueda viajar por la red a través de un socket, cada parte del mensaje debe estar correctamente alineada en su byte correspondiente. La

estructura en sí no puede transmitirse directamente; debe ser **serializada** en un array de bytes mediante la función `dump`. Este proceso permite que el mensaje se convierta en una secuencia continua de bytes que pueda viajar a través del socket.



Por otro lado, al recibir el paquete, es necesario reconstruir esa secuencia de bytes de vuelta en su forma original. Aquí entra en juego la función `load`, que toma ese array de bytes y lo **deserializa**, es decir, lo interpreta y lo convierte nuevamente en la estructura original para que pueda ser manipulada de manera sencilla en el código. Es un ida y vuelta necesario, ya que aunque la estructura sea intuitiva y fácil de usar en el código, la red necesita que todo esté bien empaquetado, byte por byte, para poder transmitirlo correctamente.

```go
func hdrLoad(b []byte) (*hdr, error) {
    if len(b) < hdrsz {
        return nil, fmt.Errorf("invalid buffer size")
    }
    crc := crc32.ChecksumIEEE(b)
    h := &hdr{
        ver:   b[0],
        kind:  b[1],
        len:   binary.BigEndian.Uint16(b[2:]),
        src:   binary.BigEndian.Uint16(b[4:]),
        dst:   binary.BigEndian.Uint16(b[6:]),
        epoch: binary.BigEndian.Uint32(b[8:]),
        time:  binary.BigEndian.Uint64(b[12:]),
        crc32: crc,
    }
    if h.ver != protocolVersion {
        return nil, fmt.Errorf("invalid protocol")
    }
    if h.kind != typeClientHandshake &&
        h.kind != typeServerHandshake &&
        h.kind != typeCtrlMessage &&
        h.kind != typeData {
        return nil, fmt.Errorf("invalid message")
    }
    return h, nil
}

func (h *hdr) dump(b []byte) error {
    if b == nil || len(b) < hdrsz {
        return fmt.Errorf("invalid buffer size")
    }
    b[0] = h.ver
    b[1] = h.kind
    binary.BigEndian.PutUint16(b[2:], h.len)
    binary.BigEndian.PutUint16(b[4:], h.src)
    binary.BigEndian.PutUint16(b[6:], h.dst)
    binary.BigEndian.PutUint32(b[8:], h.epoch)
    binary.BigEndian.PutUint64(b[12:], h.time)
    h.crc32 = crc32.ChecksumIEEE(b[:hdrsz])
    return nil
}
```

Antes de mostrar un fragmento de código, es fundamental aclarar el **orden de los bytes**. Como en todo protocolo de red, los datos se transmiten en **BigEndian**. Esto significa que los bytes más significativos se envían primero. Esta aclaración es importante porque no todos los **CPU** organizan los datos multibyte de la misma manera. Algunas arquitecturas utilizan BigEndian, mientras que otras, como la mayoría de los procesadores modernos, utilizan **LittleEndian**. Esta diferencia en cómo se almacenan los datos en la memoria puede causar problemas si no se tiene en cuenta al implementar un protocolo de red.

Por lo tanto, al serializar y deserializar los datos, debemos asegurarnos de que el protocolo siempre utilice **BigEndian**, sin importar el tipo de CPU que esté en

uso. Este detalle garantiza que los paquetes se transmitan de manera consistente y puedan ser interpretados correctamente en cualquier dispositivo, sin importar la arquitectura. En la imagen posterior se puede apreciar como se ve el código que realiza la serialización y deserialización de la cabecera convirtiendo los datos a **BigEndian**

Como se puede apreciar en el código, la función `hdrLoad()` lee desde un buffer y convierte los datos de **BigEndian** al endian nativo del host. Luego de parsear toda la cabecera, verifica que el mensaje sea válido y que el tipo de mensaje coincida con alguno de los declarados por el protocolo. Existen 4 tipos de mensaje: ***ClientHandshake, ServerHandshake, CtrlMessage y Data***. Los tres primeros viajan en **texto plano** y están firmados con la clave privada del emisor, garantizando su autenticidad. El mensaje de tipo **Data**, en cambio, se envía encriptado utilizando el **secreto compartido** generado durante el handshake, asegurando la confidencialidad de los datos.

A continuación se muestra parte del código para el proceso de serialización-deserialización del mensaje tipo handshake. El principio es el mismo, pero se agrega la verificación de la firma.

https://github.com/emilianobilli/sudp/blob/main/hdr.go

En este caso, para la serialización del mensaje, se utiliza la clave privada para firmar y en el caso de la deserialización, la clave pública del emisor para verificar la firma

```
 9  const handshakesz = 4 + 65 + 64
10
11  type handshake struct {
12      crc32     uint32
13      pubkey    [65]byte
14      signature [64]byte
15  }
16
17  func handshakeLoad(b []byte, v *ecdsa.PublicKey) (*handshake, error) {
18      if len(b) < handshakesz {
19          return nil, fmt.Errorf("invalid buffer size")
20      }
21      hs := handshake{}
22      copy(hs.signature[:], b[4+65:handshakesz])
23      if ok := verifySignature(v, b[0:4+65], hs.signature); !ok {
24          return nil, fmt.Errorf("invalid signature")
25      }
26      hs.crc32 = binary.BigEndian.Uint32(b[0:4])
27      copy(hs.pubkey[:], b[4:4+65])
28      return &hs, nil
29  }
30
31  func (h *handshake) dump(b []byte, s *ecdsa.PrivateKey) error {
32      var e error
33      if len(b) < handshakesz {
34          return fmt.Errorf("invalid buffer size")
35      }
36      binary.BigEndian.PutUint32(b[0:4], h.crc32)
37      copy(b[4:4+65], h.pubkey[:])
38      h.signature, e = signMessage(s, b[0:4+65])
39      if e != nil {
40          return e
41      }
42      copy(b[4+65:], h.signature[:])
43      return nil
44  }
45
```

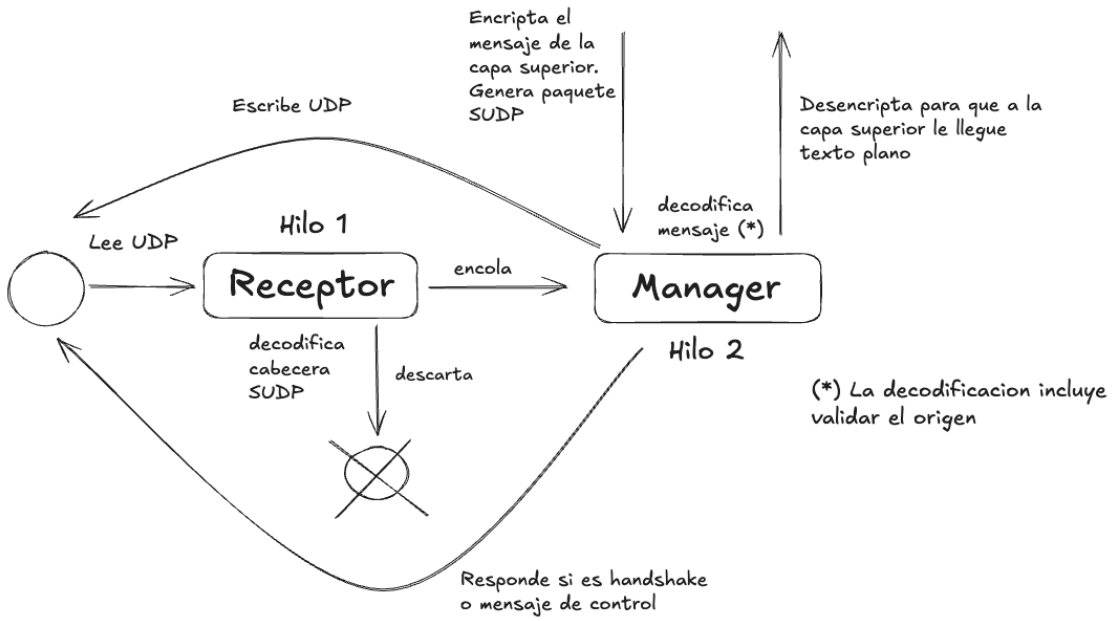
La Arquitectura

El protocolo sigue una arquitectura de **modelo cliente-servidor**, donde una parte actúa de manera completamente activa (el cliente), mientras que la otra (el servidor) es pasiva. El cliente siempre inicia la comunicación, y el servidor acepta conexiones entrantes. Un aspecto clave a tener en cuenta es que el servidor puede gestionar múltiples conexiones simultáneas, mientras que el cliente se comporta como si estuviera en modo "conexión" con un servidor específico.

El servidor solo acepta conexiones de entidades que conoce de antemano. Para verificar que "Alice" es realmente "Alice", el servidor debe poseer información sobre ella, como su **ID** y su **clave pública de firma**. Esto garantiza que solo se establezcan comunicaciones seguras y autorizadas.

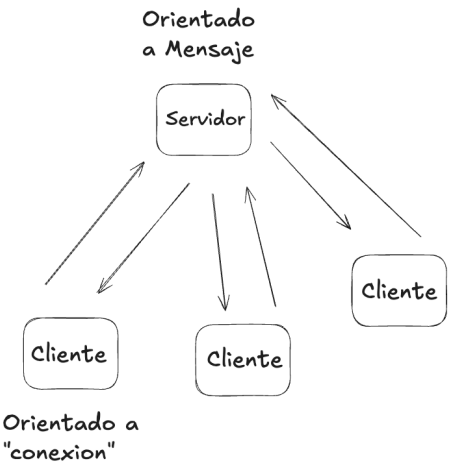
Dado que **Golang** es un lenguaje que maneja muy bien la concurrencia, tanto el cliente como el servidor operan con dos hilos de ejecución principales:

1. El primer hilo se encarga de recibir información del socket **UDP**, donde llegan los mensajes de red.
2. El segundo hilo, el principal, gestiona los mensajes que vienen tanto de la capa de "usuario" (recordemos que este protocolo también opera en la capa de usuario) como los mensajes que se reciben del primer hilo. Este segundo hilo es también responsable de enviar los mensajes UDP de forma síncrona, ya sea en respuesta a un mensaje de la capa de usuario o en el caso de mensajes de control, como un **handshake**. Además, maneja la entrega de mensajes a la capa de usuario cuando el mensaje recibido a través de UDP está dirigido a un cliente específico.



Para identificar de dónde provienen y hacia dónde van los mensajes, la cabecera incluye dos campos de 16 bits llamados **src** (origen) y **dst** (destino). En este esquema, el servidor siempre tiene el **ID 0**, mientras que los clientes se identifican con un **ID** que varía entre 1 y 65,534

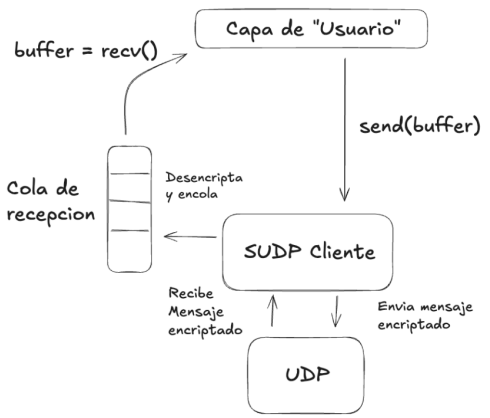
El cliente está **orientado a "conexión"**, lo que significa que solo puede recibir mensajes provenientes del servidor. En cambio, el servidor está **orientado a mensajes**, ya que puede recibir comunicaciones de cualquiera de sus clientes. Esta diferencia es fundamental, ya que el servidor, al recibir un mensaje, utiliza el campo `src` para identificar de qué cliente proviene y así poder direccionar correctamente a la capa de "usuario" o procesar la información según corresponda.

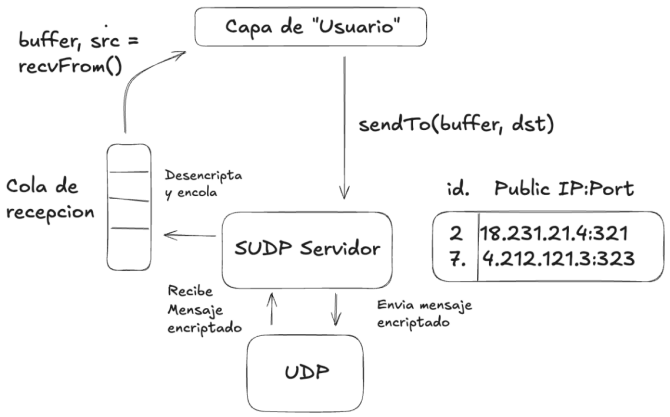


Esta arquitectura le permite al servidor gestionar múltiples conexiones y mantener las interacciones organizadas, mientras que el cliente se enfoca exclusivamente en comunicarse con su servidor asignado.

No es mi intención profundizar en cómo funciona la API **Berkeley** de sockets, pero vale la pena mencionar cómo se manejan las conexiones en ambos extremos. En el lado del **cliente**, que utiliza una "conexión" (parecida a **TCP)**, los mensajes que recibe siempre provienen del servidor. Esto se traduce en las llamadas típicas de `Connect`, `Send`, y `Recv`.

Por otro lado, el **servidor** maneja los mensajes como si estuviera utilizando **UDP**, lo que significa que puede recibir datos desde cualquier origen. Aquí, las llamadas relevantes son `Listen`, `RecvFrom`, y `SendTo`, lo que le permite gestionar múltiples clientes de manera simultánea y recibir mensajes de cualquiera de ellos sin necesidad de una conexión establecida de manera formal como en TCP.





Recapitulando

SUDP es un protocolo de transporte seguro que añade mecanismos de autenticación y encriptación, pero no garantiza ni el orden ni la recepción de los mensajes. Es decir, si un mensaje se pierde, a SUDP **no le importa**: se comporta casi como un mensaje IP o UDP tradicional. La única diferencia significativa es que SUDP añade una capa de seguridad que incluye autenticidad y confidencialidad. Aunque la arquitectura puede parecer un poco más compleja debido a su implementación en la **capa de usuario**, esta elección es necesaria para lograr el nivel de control y seguridad que ofrece el protocolo.

Puntos clave hasta ahora:

- Modelo cliente-servidor:
    - El cliente es activo y orientado a conexión (inicia la comunicación).
    - El servidor es pasivo y orientado a mensajes (puede recibir comunicaciones de múltiples clientes).
- Handshake de tres vías:
    - Permite que ambas partes se aseguren de que están listas para comunicarse.
    - Incluye el intercambio de claves públicas efímeras mediante **Diffie-Hellman**, necesario para generar un secreto compartido y encriptar los paquetes IP.
- Mensaje de datos:
    - Utiliza encriptación AES-GSM e incluye información del header (su hash)
- Mensajes de control:
    - **Keepalive**: enviado periódicamente para mantener la conexión activa, especialmente útil detrás de un NAT.
    - **Confirmación de epoch**: parte del handshake que se repite durante la comunicación, generando nuevas claves en cada epoch.
- No garantiza el orden ni la recepción:
    - SUDP se comporta como UDP, sin preocuparse por la pérdida de mensajes. Se enfoca en la **seguridad**, no en la fiabilidad del transporte.
- Arquitectura concurrente en Golang:
    - Tanto cliente como servidor tienen hilos dedicados para gestionar la recepción y envío de mensajes de forma eficiente.

Bueno, hasta aquí he llegado tras escribir y reescribir la especificación del protocolo unas tres o cuatro veces, y después de programarlo y refactorizar una gran cantidad de veces. Ahora tengo un **módulo/librería** que me permite intercambiar mensajes de manera segura entre un cliente y un servidor. Desde el lado del cliente, puedo conectarme y enviar mensajes cifrados, y desde el lado del servidor, puedo recibir mensajes seguros de cualquier cliente que tenga "registrado".

Hasta este punto, la parte del protocolo cubre tres aspectos fundamentales: **transporte**, **autenticación** y **encriptación**. Sin embargo, ahora toca avanzar un poco más hacia la **capa de red**. Mi próximo desafío es capturar paquetes IP, encapsularlos dentro de **SUDP**, enviarlos a través de la red, y finalmente, en el otro extremo, recibir esos paquetes e **inyectarlos** nuevamente en la capa de red.

Parte 2. La capa de Red

Al empezar a trabajar en la **capa de red**, me di cuenta de que era esencial capturar y manipular los **paquetes IP** de una manera eficiente, pero no bastaba con simplemente usar **IPPROTO_RAW** para enviar y recibir paquetes crudos. Para que todo funcionara como debía, necesitaba una **interfaz de red** configurada, y aquí es donde la cosa se pone interesante.

¿Por qué una interfaz de red?

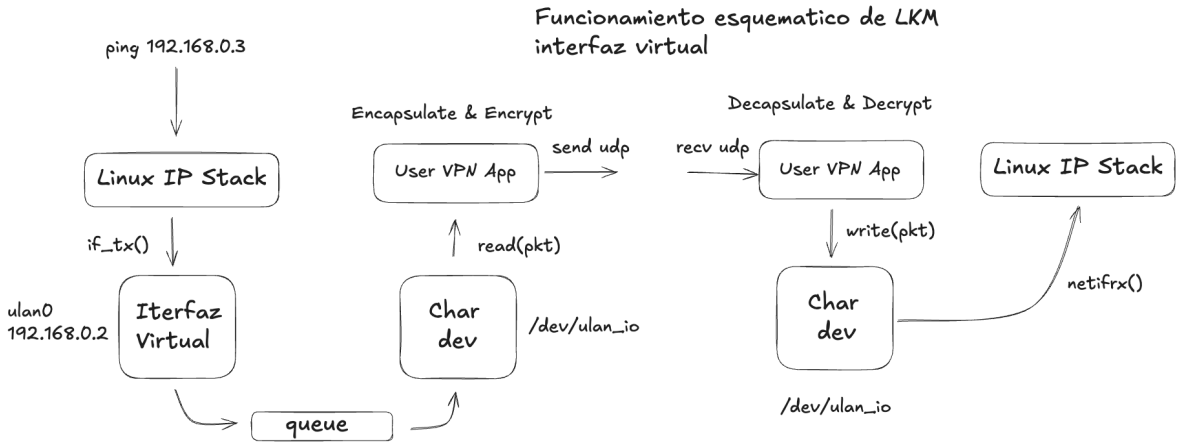
Al principio, puede parecer que bastaría con capturar paquetes crudos directamente, pero la realidad es que una interfaz de red hace mucho más que eso. No solo permite enviar y recibir paquetes, también juega un papel crucial en el **enrutamiento**. Si solo enviara paquetes con **IPPROTO_RAW**, el sistema no tendría idea de por dónde enrutarlos, y todo se complicaría.

Luego está el tema de la **IP asignada** a la interfaz. Sin una IP configurada, el sistema no sabría qué paquetes procesar o a qué tráfico responder. Básicamente, la interfaz se convierte en el punto de referencia para que el sistema sepa qué paquetes debe gestionar.

Además, la interfaz filtra y organiza el tráfico, algo que con **IPPROTO_RAW** sería mucho más complicado. En lugar de recibir solo el tráfico que me interesa, recibiría todo el tráfico crudo, y sería un caos gestionarlo.

Mi enfoque inicial

Cuando empecé a pensar en cómo capturar paquetes, mi primera idea fue escribir un **driver del kernel de Linux**. La idea era crear una **interfaz virtual** que actuara como una tarjeta de red dummy y luego desarrollar un **driver de carácter** que permitiera leer y escribir datos en esa interfaz. Todo funcionaba bien... hasta que actualicé al **kernel 6**. Ahí empezaron los problemas de **compatibilidad** y me di cuenta de que mi enfoque, además de ser poco flexible, estaba limitando la **portabilidad** de mi solución.



Descubriendo TUN/TAP

Fue entonces cuando descubrí los módulos **TUN/TAP**. Estos módulos me permitieron crear **interfaces virtuales** desde el espacio de usuario, sin necesidad de tocar el kernel. **TUN** maneja paquetes IP a nivel de red, mientras que **TAP** se encarga de los paquetes a nivel de enlace. Era una solución elegante que me permitía inyectar y capturar paquetes de una manera mucho más sencilla, y lo mejor de todo, **portable**.

Con **TUN/TAP**, pude integrar todo sin preocuparme de los problemas de compatibilidad y sin necesidad de mantener un driver del kernel personalizado. Además, estas interfaces virtuales se integran perfectamente con la pila de red, permitiendo que todo funcione como si fuera una tarjeta de red real, pero sin la complicación de manejar el hardware.

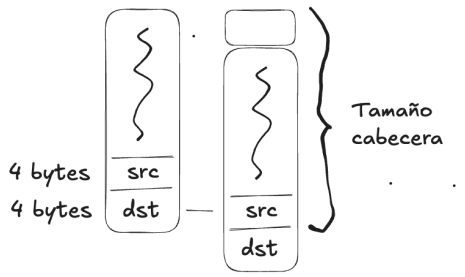
Y en macOS...

Pero no todo era Linux. Como también trabajo en **macOS**, tenía que encontrar una solución equivalente allí. Fue entonces cuando descubrí **utun**, la versión de **TUN** en macOS. **utun** me permite hacer exactamente lo mismo que **TUN** en Linux: crear una **interfaz virtual** que me permite capturar y enviar paquetes IP directamente desde el espacio de usuario.

Al usar **utun** en macOS y **TUN/TAP** en Linux, logré que mi solución fuera totalmente **portable** entre ambos sistemas operativos. Ahora puedo capturar, manipular y reenviar paquetes en cualquier plataforma, sin preocuparme por las diferencias entre ellas. La portabilidad y la flexibilidad se convirtieron en los pilares de mi implementación.

Hasta ahora, en teoría, todo marchaba bien. Sin embargo, al avanzar en la lectura de paquetes IP, me encontré con un problema inesperado: el dispositivo de red **utun** me estaba entregando **cabeceras IP incorrectas**. Había una especie de desplazamiento en los datos, y me costó bastante darme cuenta de lo que estaba pasando, hasta que finalmente algo hizo que se me encendiera la lamparita.

Para entender qué ocurría, decidí hacer un **dump** de todos los paquetes IP. Y ahí lo vi: en el campo del encabezado donde debería estar la **dirección de destino** del paquete IP, estaba la **dirección de origen**. Al principio parecía que no había ninguna alteración, pero al observar con más detalle, me di cuenta de que había un **desplazamiento de 4 bytes** al comienzo del paquete.



Y aquí está lo curioso: ese desplazamiento de 4 bytes era justo lo que permitía que la dirección de origen se colocara en el campo de destino. Si el desplazamiento hubiera sido diferente, probablemente habría sido mucho más difícil de detectar, pero al ser exactamente de 4 bytes, las direcciones simplemente cambiaban de lugar, lo que hizo posible identificar el problema.

Al parecer, a los genios de BSD/Darwin se les ocurrió agregar 4 bytes al principio de cada paquete que debía entregarse a la capa superior, y lo mismo ocurría al escribir un paquete: también había que agregar esos 4 bytes al inicio. Lo más gracioso de todo es que esto no está documentado en ningún lugar, lo que me hizo perder un montón de tiempo intentando descubrir qué estaba pasando. Es realmente frustrante cuando detalles como este no están explicados en ninguna parte.

La semántica de estos 4 bytes es la siguiente:

- Los primeros dos bytes son reservados y generalmente tienen un valor de 0.
- El tercer byte también es reservado, pero puede ser utilizado en futuras versiones.
- El cuarto byte es el que determina si el paquete es IPv4 o IPv6.

```go
func (u *Utun) Read(buf []byte) (int, error) {
	var tmp [2048]byte

	if len(buf) < u.MTU {
		return 0, fmt.Errorf("invalid buf len, less than MTU")
	}
	for {
		n, e := u.file.Read(tmp[:u.MTU+4])
		if e != nil {
			return 0, e
		}
		if n > 4 {
			if tmp[3] == unix.AF_INET {
				copy(buf, tmp[4:n])
				return n - 4, nil
			}
		}
	}
}

func (u *Utun) Write(buf []byte) (int, error) {
	var tmp [2048]byte
	if len(buf) > u.MTU {
		return 0, fmt.Errorf("invalid buf len, greather than MTU")
	}
	tmp[0] = 0x00
	tmp[1] = 0x00
	tmp[2] = 0x00
	tmp[3] = unix.AF_INET
	copy(tmp[4:], buf)
	return u.file.Write(tmp[0 : len(buf)+4])
}
```

Esta información es importante para todos aquellos que estén desarrollando algo parecido y, por casualidad, lean este relato. Si no sabes que estos 4 bytes están ahí, podrías quedarte atrapado en un ciclo de errores incomprensibles, como me pasó a mí. Así que, si ves un desplazamiento de 4 bytes al leer o escribir en la interfaz **utun**, ya sabes de dónde proviene el problema y cómo interpretarlo.

Pero la cosa no terminaba ahí. Una vez que solucioné el problema del desplazamiento de los 4 bytes, me encontré con otro error, o al menos una inconsistencia en cómo se mostraba la información sobre la **longitud de la cabecera IP**. Estaba enviando un **ping** desde un servidor que tengo en la nube, y la longitud del mensaje que me devolvía era de **21,504 bytes** (o algo por el estilo). Eso no tenía ningún sentido: no podía ser que el mensaje fuera tan grande, ya que el **MTU** estaba configurado con un valor mucho menor y, además, ¡era solo un ping!

Toda la demás información parecía correcta, y estaba utilizando la **librería estándar** para hacer el parseo de la cabecera IP. Fue entonces cuando se encendió **otra lamparita**: debía haber un bug en la librería estándar. No había otra explicación. Pero, ¿por qué?

Paso a explicar: el servidor en la nube es una máquina **Intel**, lo que significa que utiliza el formato **Little Endian**. Mientras tanto, mi computadora local es **Big Endian**. La longitud del mensaje se estaba **convirtiendo de forma incorrecta** debido a esta diferencia de endianess. El número que me estaba mostrando, **21,504**, es en realidad lo que resulta cuando se hace mal la conversión. La longitud real era **84 bytes**, que es exactamente lo que debería haber sido para un ping. Así que, dicho y hecho: había un **bug** en la librería estándar justo en ese punto.

Lo más frustrante es que, al investigar un poco más, encontré que este bug ya había sido reportado. Al parecer, sigue abierto desde hace más de un año... y nadie lo ha atendido todavía. Si a alguien le interesa: (https://github.com/golang/go/issues/32118) El hecho es que también está mal redactado, ya que puso amd64 y en realidad es arm64.

Y en Windows…

La verdad no me interesa.