PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
ESCOLA POLITÉCNICA

PROTOCOLO QUIC PARA SECURIZAR A CONEXÃO DE ROTEADORES

IVAN PALADIN JUNIOR

Trabalho de Conclusão de Curso apresentado como requisito parcial à obtenção do grau de Bacharel em Engenharia de Computação na Pontifícia Universidade Católica do Rio Grande do Sul.

Orientador: Prof. Fernando Gehm Moraes

Porto Alegre
2025

AGRADECIMENTOS

Gostaria de deixar aqui minha gratidão às pessoas que contribuíram ou estavam presentes durante meu caminho, desde o início de minha graduação até o desenvolvimento deste TCC.

Agradeço inicialmente à minha família, pelo apoio incondicional nos momentos bons e ruins, sempre presentes, nunca distantes. Em especial à minha avó Teresa, que não irá conseguir ler este documento, mas seu apoio desde o início, a não desistir dos meus sonhos, me permitiu chegar até esse momento.

Gostaria de agradecer ao meu orientador, Fernando Gehm Moraes, pelo apoio, pelas oportunidades de aprendizado apresentadas, paciência e inúmeros conselhos, os quais levarei para minha futura carreira como engenheiro e profissional.

Agradeço aos meus colegas do projeto SIGMARR da DATACOM/PUCRS, passados e presentes,em especial meu amigo William Damasceno, que junto com os outros tiveram um papel principal no desenvolvimento do projeto que seria meu TCC; sem vocês, isso não seria possível.

Agradeço à PUCRS, a todos os meus professores e professoras, que me acompanharam durante todos esses anos. Todos sempre foram muito atenciosos e pacientes.

E por fim, gostaria de agradecer a todos os meus amigos e colegas que tive durante todos esses anos no curso de Engenharia da Computação, que compartilharam: histórias, conhecimento e noites sem dormir fazendo trabalhos. Que vocês também cheguem a essa etapa.

PROTOCOLO QUIC PARA SECURIZAR A CONEXÃO DE ROTEADORES

RESUMO

Este trabalho de conclusão de curso (TCC) aborda o desenvolvimento de uma aplicação cujo objetivo é implementar o protocolo de transporte QUIC, em substituição ao método padrão TCP+TLS na conexão entre uma aplicação móvel e um roteador, permitindo que o usuário realize, remotamente, alterações no roteador. O problema central abordado é a necessidade de aprimorar a aplicação, visando aumentar a eficiência, o desempenho e a segurança durante o estabelecimento da conexão. O objetivo estratégico deste trabalho foi compreender o protocolo QUIC, e, a partir desse conhecimento, implementá-lo em código C. Após a realização de testes, buscou-se integrar essa implementação ao código *chownat*, utilizado como primeira versão para o estabelecimento da conexão. O *chownat* foi projetado para estabelecer uma conexão TCP encapsulada por uma conexão UDP, visando atravessar o NAT do dispositivo e permitir a realização de alterações sem que o cliente e o roteador precisem estar conectados à mesma rede. Neste contexto, o trabalho definiu dois objetivos de desenvolvimento. O primeiro: implementar o protocolo QUIC em uma aplicação simples, com o propósito de aplicar os conhecimentos adquiridos durante a pesquisa. O segundo: realizar a integração parcial do código QUIC desenvolvido ao código *chownat*. A seção de resultados apresenta o funcionamento da implementação, bem como o resultado da integração dos códigos.

Palavras-Chave: Conexão remota, QUIC, chownat, Roteadores, Linguagem C.

QUIC PROTOCOL TO SECURE ROUTER CONNECTIONS

ABSTRACT

This undergraduate thesis addresses the development of an application that implements the QUIC transport protocol to replace the standard TCP+TLS method in the connection between a mobile application and a router, thereby enabling users to modify router settings remotely. The central problem examined is the need to enhance the application to increase efficiency, performance, and security during connection establishment. The strategic objective of this work was to study the QUIC protocol and, based on this understanding, implement it in C code. Following the implementation and testing phases, an attempt was made to integrate this implementation into the *chownat* code, which served as the initial version for connection establishment. The *chownat* tool was designed to establish a TCP connection encapsulated within a UDP connection, allowing traversal of the device's NAT and enabling modifications without requiring the client and router to be on the same network. In this context, the project defined two objectives. The first was to implement the QUIC protocol in a simple application, with the goal of applying the knowledge acquired during the research phase. The second was to carry out the partial integration of the developed QUIC code into the *chownat* codebase. The results section presents the functioning of the implementation as well as the outcomes of the code integration.

Keywords: Remote Connection, QUIC, chownat, routers , C language.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1. INTRODUÇÃO

Diante da crescente evolução tecnológica dos dispositivos móveis nos últimos 30 anos, observa-se a necessidade de expandir o acesso à informação e serviços para os diversos públicos que compõem esse mercado. Essa expansão ocorre por meio do desenvolvimento de novas aplicações para dispositivos móveis, que podem ser utilizadas em qualquer lugar e a qualquer momento. Essas aplicações têm como foco principal a rapidez tanto na solicitação quanto na resposta dos serviços solicitados pelo usuário, a acessibilidade desses serviços ao público e a segurança frente a possíveis usuários mal-intencionados.

Com isso em mente, houve uma intensificação no desenvolvimento de aplicações para dispositivos móveis, o que fez com que sua usabilidade deixasse de se restringir ao uso pessoal, passando também a atender a demandas profissionais em diversas áreas, como administração, educação, saúde, comunicação, entre outras.

Diante desse cenário, a empresa de tecnologia e comunicação DATACOM, em parceria com a PUCRS, propôs um projeto com o objetivo principal de desenvolver uma aplicação para dispositivos móveis que permitisse a configuração local e remota das diferentes famílias de roteadores comercializadas pela DATACOM, com a finalidade inicial de facilitar o trabalho de seus técnicos na manutenção dos equipamentos e, futuramente, disponibilizar essa funcionalidade para seus clientes.

Adicionalmente, foi sugerido que o desenvolvimento da aplicação incorporasse melhorias, especialmente na construção dos métodos implementados, buscando incluir inovação no *software* de gerência dos roteadores. Um dos métodos desenvolvidos para aprimorar a qualidade do serviço foi a possibilidade de alterar os parâmetros de um roteador da DATACOM remotamente, sem a necessidade de estar conectado à mesma rede do dispositivo a ser configurado.

No entanto, essa implementação apresenta vulnerabilidades: o conteúdo dos pacotes transmitidos durante a comunicação entre o roteador e o *smartphone* pode ser interceptado por qualquer usuário com acesso a um *software* de monitoramento de rede. Para mitigar essa vulnerabilidade, decidiu-se pela implementação do protocolo de transporte QUIC [13, 15], que, além de oferecer maior velocidade em comparação com o protocolo *TLS (Transport Layer Security)* [8], também apresenta maior segurança em relação ao protocolo *TCP (Transmission Control Protocol)* [4].

1.1 Motivação

Diante da necessidade de implementar uma solução que garantisse a confidencialidade das informações transmitidas entre o roteador e a aplicação, foi realizada uma

pesquisa sobre as vantagens e desvantagens da utilização dos protocolos padronizados *TCP+TLS* e QUIC.

Após uma análise de ambos os protocolos, optou-se pela utilização do QUIC, devido ao aumento da segurança proporcionada. Embora utilize o mesmo conceito de uma conexão *TCP+TLS*, o QUIC apresenta a vantagem de ter o protocolo de segurança *TLS* implementado diretamente em sua estrutura, eliminando a necessidade de utilizá-lo como uma extensão, como é feito normalmente, porém aumentando a segurança dos pacotes que utilizam o protocolo e reduzindo a latência nas etapas de autenticação de conexão.

Além da análise inicial efetuada, também foi considerado que o protocolo foi projetado para ser uma alternativa moderna ao modelo *TCP+TLS*. Já que, desde sua criação, houve um aumento significativo de seu uso em relação ao modelo antigo, sendo muito utilizado por sites de alto tráfego de dados, como Instagram, Google e Facebook, chegando a ser usado por 8,6% [19] de todos os sites da internet.

1.2 Objetivos

O objetivo **estratégico** deste TCC é implementar o protocolo QUIC para a comunicação remota entre o aplicativo e o roteador, com foco na autenticação da conexão e na troca de informações entre ambos, garantindo a segurança, integridade, confidencialidade e disponibilidade dos dados. O aplicativo mencionado acima é executado em um *smartphone*, denominado pelo nome do projeto em parceria com a DATACOM: **SIGMARR**, uma sigla para: "**S**olução **I**ntegrada de **G**erenciamento e **M**onitoramento de **R**edes **R**esidenciais".

Para alcançar esse objetivo, foram definidos os seguintes objetivos **específicos**:

1. Compreender o funcionamento do protocolo QUIC, tanto de forma geral quanto no nível detalhado. Esta etapa abrange o estudo da estrutura do protocolo, incluindo alocação, *handshakes* iniciais e os protocolos e algoritmos de segurança utilizados para a criptografia dos dados.
2. Entender a implementação que realiza a conexão remota entre o roteador e a aplicação SIGMARR. O programa que realiza esta conexão remota é o *CHOWNAT* [17]. Esta compreensão é essencial para permitir a integração do protocolo QUIC ao código que realiza a conexão remota.
3. Desenvolver uma implementação prova-de-conceito do protocolo QUIC, utilizando bibliotecas que já contenham as funcionalidades do protocolo prontas, visando avaliar o funcionamento de suas diferentes funções e o comportamento da implementação desenvolvida, para que sejam utilizadas no desenvolvimento da aplicação final do projeto.
4. Iniciar o desenvolvimento da aplicação, utilizando a biblioteca que demonstrou um melhor retorno do protocolo QUIC, com foco na implementação e integração das lógicas

de autenticação e transferência de dados utilizadas pelo protocolo. Também será realizada, a implementação das funções de criptografia (*AES-128* [7]) e dos algoritmos correspondentes (*RSA* [12]), visando garantir a confidencialidade das informações trocadas. Essas escolhas de função de criptografia e algoritmo foram decididas pelo autor deste documento. Outros tipos de funções e algoritmos são suportados pelo protocolo, tendo a opção de serem utilizadas, por exemplo: *AES-256* e *ECDHE*.

5. Realização de testes, que consistem em uma comunicação entre dois dispositivos denominados *Cliente* e *Servidor*. A validação será feita por meio do *software* de monitoramento de rede *Wireshark* [18]. O objetivo é confirmar que a comunicação e a criptografia dos dados estão de acordo com as especificações do protocolo QUIC.

1.3 Organização do Documento

Este TCC é organizado como segue:

- O Capítulo 2 apresenta os conceitos básicos necessários à compreensão deste trabalho.
- O Capítulo 3 detalha o protocolo QUIC, utilizado no desenvolvimento desse TCC.
- O Capítulo 4 detalha a implementação do protocolo utilizando a biblioteca *MsQuic*.
- O Capítulo 5 apresenta resultados da implementação.
- O Capítulo 6 conclui este trabalho e aponta direções para trabalhos futuros.

2. CONCEITOS BÁSICOS RELACIONADOS A PROTOCOLOS DE REDE E ALGORITMOS DE CRIPTOGRAFIA

Este capítulo apresenta conceitos básicos necessários para o entendimento dos temas estudados e implementados nesse trabalho de conclusão. A Figura 2.1 apresenta os atores envolvidos na comunicação e os protocolos utilizados (representados nas setas).

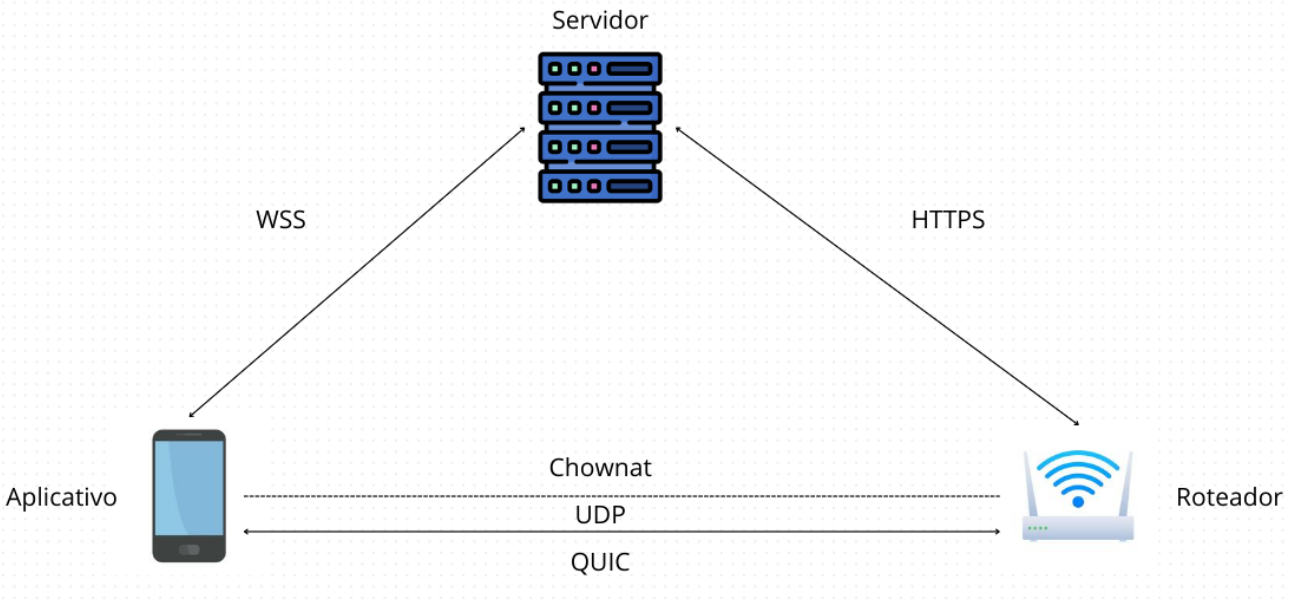

Figura 2.1 – Arquitetura de comunicação utilizada neste TCC.

Este capítulo está organizado em cinco seções:

- Seção 2.1: apresenta os protocolos de transporte, *UDP* e o *TCP*;
- Seção 2.2: apresenta o programa CHOWNAT, necessário para realizar conexões remotas através do *NAT*;
- Seção 2.3: apresenta o protocolo *TLS*, necessário para a comunicação segura entre clientes e servidores;
- Seção 2.4: detalha conceitos de criptografia simétrica e assimétrica, necessários para os protocolos utilizados neste trabalho;
- Seção 2.5: apresenta os protocolos de aplicação, notadamente o HTTP, no qual o protocolo QUIC se insere.

2.1 Protocolos de Transporte

Esta seção apresenta conceitos básicos relacionados aos protocolos de transporte *UDP* e *TCP*.

2.1.1 UDP

O *User Datagram Protocol* (*UDP*) é um protocolo de comunicação focado na redução da latência no recebimento de informações, sem garantir a entrega, a ordem ou a confiabilidade dos pacotes de dados. Isso é possível devido à ausência de uma pré-conexão com o dispositivo de destino e ao tamanho reduzido de seu cabeçalho (8 bytes, enquanto seu protocolo correspondente, o *TCP*, utiliza de 20 a 60 bytes). Por esses motivos, o *UDP* é ideal para aplicações como videochamadas, jogos online, entre outras.

A Figura 2.2 apresenta as fases presentes na comunicação *UDP* entre o ator que envia dados (*Sender*) e o receptor (*Receiver*). Utilizando o contexto do projeto citado no Capítulo 1, o receptor é o cliente realizando uma solicitação de informações específicas sobre o *Sender*, que, no caso, é o roteador e o mesmo o responde enviando de volta pacotes que contêm a informação solicitada. Resumidamente, o protocolo permite que o roteador possa simplesmente começar a enviar dados para o cliente sem estabelecer uma conexão prévia.

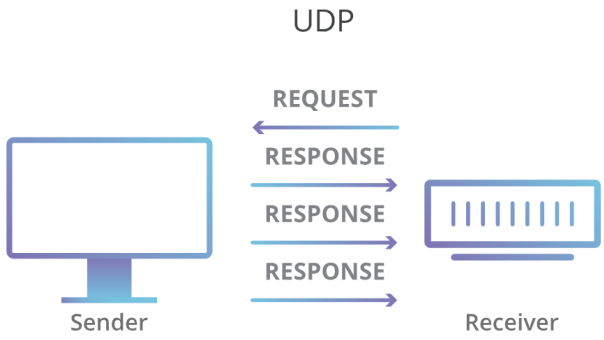

Figura 2.2 – Fases do protocolo UDP (Fonte: [2]).

2.1.2 TCP

O *Transmission Control Protocol* (*TCP*) é um protocolo de comunicação com funcionamento distinto ao do *UDP*, por visar garantir a entrega, a ordem e a integridade das informações transmitidas. Isso é viabilizado por meio de uma pré-conexão com o dispositivo de destino, realizada mediante um processo denominado *3-way handshake*. Uma vez estabelecida essa conexão, durante a transmissão dos pacotes de dados, eventuais erros ou pacotes entregues fora de ordem podem ser detectados e corrigidos, assegurando a confiabilidade da comunicação. Esse protocolo é adequado para aplicações onde a entrega correta das informações é prioritária em relação à latência, como em e-mails, acesso a sites e transferência de arquivos.

A Figura 2.3 apresenta as fases da comunicação *TCP* entre o ator que envia dados (*Sender*) e o receptor (*Receiver*). As principais fases do protocolo incluem:

- **SYN:** O cliente (*Sender*) envia um pacote com a mensagem "*SYN*" ao servidor (*Receiver*), indicando que deseja iniciar uma conexão.
- **SYN-ACK:** O servidor responde com um "*SYN-ACK*", indicando que recebeu o pedido e está pronto para estabelecer a conexão.
- **ACK:** Ao receber a resposta do servidor, o cliente envia uma mensagem "*ACK*", confirmando o recebimento e finalizando o estabelecimento da conexão.

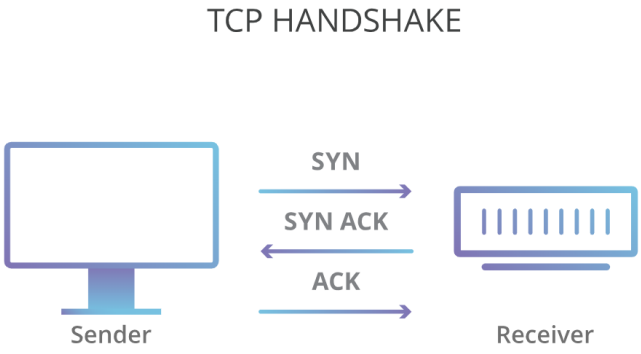

Figura 2.3 – Fases do protocolo *TCP* (Fonte: [2]).

2.2 CHOWNAT

O programa denominado CHOWNAT é desenvolvido em linguagem C e permite a comunicação entre dois *peers* localizados atrás de *NATs* distintos, sem a necessidade de utilizar *port forwarding* ou *DMZ* nos roteadores utilizados por esses *peers*. Isso é possível porque o programa estabelece um túnel entre os dois dispositivos, utilizando uma técnica chamada *hole punching*, que permite que um dos *peers* acesse um serviço — como, por exemplo, um *web server* — por meio desse túnel, atravessando o *NAT*.

No contexto deste TCC, o CHOWNAT possibilita a configuração remota de roteadores da DATACOM através do aplicativo SIGMARR, sem a necessidade de acesso convencional por navegador, inserindo o IP designado ao roteador. A ordenação do processo nesta etapa baseia-se em inteiros que representam *IDs* de cinco tipos de pacotes que podem ser enviados/recebidos nessa comunicação entre cliente e servidor:

- **01**: *ID* que sinaliza a tentativa de conexão com o servidor por parte do cliente. No servidor, esse *ID* é utilizado como parâmetro para validar que a conexão recebida é proveniente do cliente correto e, assim, estabelecer a conexão de fato.

- **02**: *ID* que sinaliza a tentativa de desconexão com o servidor por parte do cliente. No servidor, esse *ID* é utilizado como parâmetro para validar que a solicitação de desconexão recebida é proveniente do cliente correto e, assim, encerrar a conexão de fato.
- **03**: *ID* que sinaliza, no cliente, que a conexão com o servidor foi estabelecida com sucesso. No servidor, esse *ID* é utilizado para informar ao cliente que a conexão remota foi realizada com êxito, especificamente no processo de tunelamento (encapsulamento da conexão *TCP* dentro da *UDP*).
- **08**: No cliente, essa *ID* tem por finalidade informar ao servidor que o pacote recebido está incorreto, solicitando o reenvio do pacote correto. No servidor, essa *ID* é utilizada para retransmitir ao cliente o pacote solicitado corretamente.
- **09**: No cliente, essa *ID* valida se o pacote recebido é o correto (neste caso, um pacote da conexão *UDP*) e o redireciona para a porta local com a conexão *TCP* aberta, realizando assim o tunelamento da conexão e o envio do dado ao servidor. No servidor, ocorre apenas o recebimento da mensagem enviada pelo cliente e o envio da confirmação de recebimento dessa mensagem.

O programa não encerra nenhuma das duas conexões, utilizando um mecanismo de *keep-alive* que impede o *timeout*, com intervalo de 3 segundos. O programa é dividido em duas partes: cliente e servidor, resultando em diferenças na utilização dos *IDs* — por exemplo, o pacote 01 no cliente é enviado somente por ele para iniciar a comunicação, sendo utilizado como parâmetro de início de conexão no servidor. Os dois programas serão brevemente explicados a seguir, estando disponíveis em https://github.com/ipaladinjr/Chownat_TCC.

2.2.1 **chownat server.c**

O programa “chownat server.c” inicialmente abre um *socket* que aguarda a conexão do dispositivo especificado pela variável de endereço *IP* remoto. Quando for detectada a tentativa de conexão por parte do dispositivo, é estabelecida a conexão entre o servidor e o cliente, visando mantê-la ativa por meio de pacotes *keep-alive*. Essa etapa é realizada utilizando o protocolo *UDP*. Após essa fase, o programa utiliza a conexão *UDP* estabelecida para iniciar uma nova conexão, desta vez via *TCP*, com o cliente, efetuando o tunelamento através do *NAT*, ou seja, encapsulando a conexão *TCP* na conexão *UDP*, permitindo a troca de dados via *TCP* por meio do *NAT*.

2.2.2 **chownat client.c**

Já no cliente, é ele quem inicia efetivamente a conexão com o servidor. No contexto do projeto, o cliente corresponde à aplicação móvel que tenta se conectar ao servidor (roteador). A princípio, sua codificação é semelhante à do *server*, com a diferença de que o cliente não abre um *socket* em uma porta específica nem permanece em estado de *listening*, aguardando novas conexões. Em vez disso, ele inicia a conexão utilizando a porta e o *IP* especificados no *server* como parâmetros.

2.3 TLS

O *Transport Layer Security* (*TLS*) é um protocolo criptográfico que atua em conjunto com o protocolo *HTTP*. Foi desenvolvido inicialmente em 1994, com sua primeira publicação ocorrendo em 1999, como uma alternativa ao seu antecessor, o *SSL* (*Secure Sockets Layer*). Seu objetivo principal é assegurar a confidencialidade dos dados durante a comunicação entre dispositivos, especialmente na autenticação de conexões de rede e nos processos de criptografia e descriptografia durante a transmissão de dados.

O *TLS* busca garantir três funções principais:

- Criptografia: assegura que os dados transferidos permaneçam ocultos para terceiros;
- Autenticação: garante que as partes envolvidas na troca de informações sejam autênticas;
- Integridade: verifica se os dados não foram falsificados ou adulterados durante a transferência.

O *TLS* é amplamente utilizado na comunicação entre aplicativos web e servidores, como na navegação por sites, mas também pode ser aplicado à criptografia de outras formas de comunicação, como e-mails, mensagens e VoIP (*Voice over IP*).

Uma de suas contribuições mais conhecidas é a construção do protocolo *HTTPS*, que corresponde à implementação do *TLS* sobre o protocolo *HTTP/2*, o qual será abordado ao final deste capítulo. O uso do *HTTPS* é atualmente considerado indispensável pelos navegadores.

O funcionamento do *TLS* inicia-se por meio de uma série de etapas conhecidas como *TLS handshake*. Quando um usuário acessa um site que utiliza *TLS*, o *handshake* é iniciado entre o dispositivo do usuário (também chamado de cliente) e o servidor web. Utilizando a Figura 2.4 como referência, são descritas a seguir as etapas e operações realizadas durante essa comunicação entre cliente e servidor:

- Em “ClientHello”, o cliente inicia o *handshake* enviando uma mensagem de saudação ao servidor. Essa mensagem inclui a versão do *TLS* suportada pelo cliente (*TLS 1.0, 1.2, 1.3* etc.), os conjuntos de cifras compatíveis e uma sequência aleatória de 32 bytes, conhecida como “cliente aleatório” (*client random*). Essa sequência é gerada aleatoriamente pelo cliente e auxilia na geração de chaves de sessão seguras.
- Ao receber o “ClientHello”, o servidor responde com uma mensagem “ServerHello”. Nessa resposta, o servidor envia seu certificado *SSL*, o conjunto de cifras selecionado e uma nova sequência aleatória de 32 bytes, chamada “servidor aleatório” (*server random*), gerada pelo servidor.
- O cliente, ao receber a resposta, verifica o certificado *SSL* do servidor junto à autoridade certificadora (*CA*) que o emitiu. Isso confirma a identidade do servidor e assegura que o cliente está se comunicando com o legítimo proprietário do domínio.
- Em seguida, o cliente envia uma nova sequência aleatória de bytes, chamada “segredo pré-mestre” (*pre-master secret*). Esse valor é criptografado com a chave pública do servidor, obtida a partir do certificado *SSL*, e só pode ser descriptografado pelo servidor com sua chave privada.
- O servidor descriptografa o segredo pré-mestre. A partir do cliente aleatório, do servidor aleatório e do segredo pré-mestre, tanto o cliente quanto o servidor geram as mesmas chaves de sessão.
- Cliente e servidor trocam mensagens de “finalizado”, ambas criptografadas com a chave de sessão, sinalizando que a negociação foi concluída com sucesso.
- O *handshake* é então finalizado, e a comunicação subsequente ocorre com o uso das chaves de sessão estabelecidas.

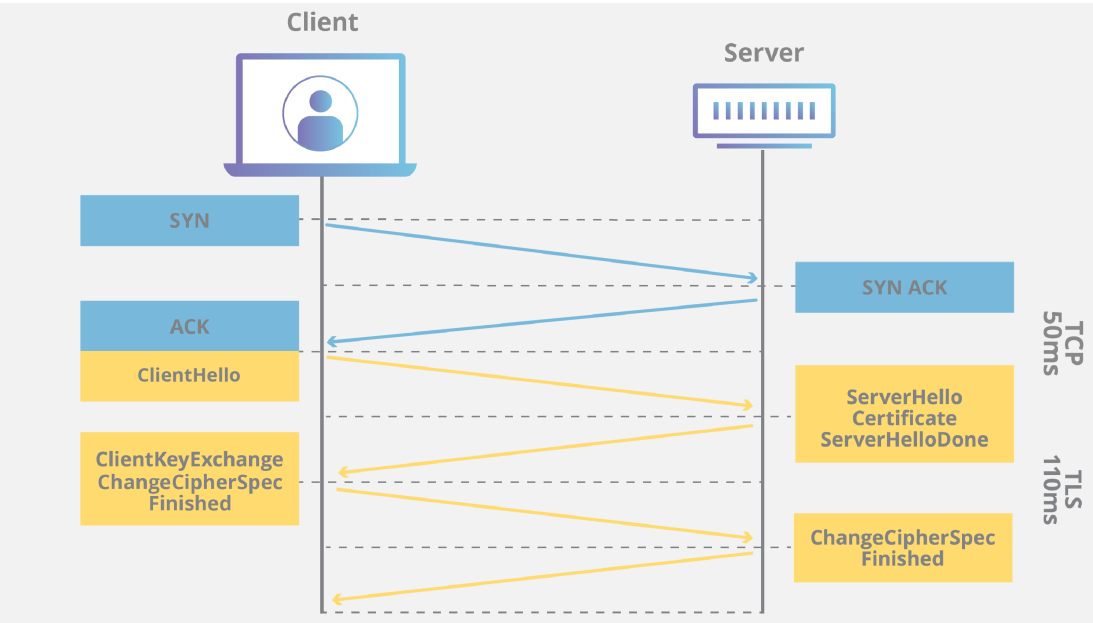

Figura 2.4 – Diagrama de Conexão TLS+TCP (Fonte: [3]).

Na etapa de criptografia, é definido um pacote de codificação para cada sessão de comunicação. Esse pacote consiste em um conjunto de algoritmos que especificam, por exemplo, quais chaves de criptografia ou chaves de sessão compartilhadas serão utilizadas em uma sessão específica. O *TLS* estabelece as chaves de sessão em um canal não criptografado por meio de criptografia de chave pública.

Na etapa de autenticação, o servidor comprova sua identidade ao cliente. Esse processo é realizado com o uso de chaves públicas, que utilizam criptografia unidirecional. Isso significa que qualquer pessoa com a chave pública pode verificar dados que foram criptografados com a chave privada do servidor, garantindo sua autenticidade. No entanto, somente o remetente original, que possui a chave privada, pode realizar a criptografia dos dados. A chave pública do servidor integra o certificado *TLS*.

Por fim, após as etapas de criptografia e autenticação, os dados são assinados com um código de autenticação de mensagem (*MAC*), permitindo ao destinatário verificar a integridade dos dados. Essa verificação assegura que o conteúdo recebido não foi alterado durante a transmissão, pois qualquer modificação nos dados resultaria em um *MAC* inválido.

2.4 Criptografia

A criptografia refere-se a um conjunto de técnicas destinadas à proteção de dados ou informações contra o acesso, visualização ou alteração por pessoas não autorizadas. O objetivo é evitar o uso malicioso dessas informações para fins como obtenção de ganho financeiro, disrupção de serviços ou estruturas no contexto em que os dados estão inseridos, ou ainda para acessar outros dados com níveis mais elevados de confidencialidade.

A criptografia tem por finalidade atender a quatro princípios fundamentais na proteção de dados:

- **Confidencialidade**: Garante que os dados sejam acessados somente pelos destinatários autorizados.
- **Integridade**: Assegura que os dados não sejam alterados durante a transmissão até o destinatário.
- **Autenticação**: Verifica a autenticidade das informações ou a identidade de um usuário, assegurando que ele tenha permissão para acessar os dados.
- **Não repúdio**: Garante que nenhuma das partes envolvidas possa negar sua participação na comunicação.

Para garantir essa proteção, os dados devem ser cifrados. A cifra consiste em um processo de codificação que torna a mensagem ilegível para terceiros. Existem diversos algoritmos e métodos para isso, sendo os principais baseados no uso de chaves. Uma

chave funciona como uma senha que permite decifrar os dados cifrados, possibilitando que somente o remetente e o destinatário autorizados possam acessá-los.

Existem dois principais tipos de chaves: chave pública e chave privada. A chave privada deve ser mantida em sigilo e não pode ser compartilhada. A chave pública, por sua vez, pode ser distribuída livremente. Com esses conceitos introdutórios sobre criptografia e seus elementos, a próxima seção aborda dois tipos principais de criptografia: simétrica e assimétrica.

2.4.1 Criptografia Assimétrica

A criptografia assimétrica, também conhecida como criptografia de chave pública, utiliza duas chaves distintas no processo: uma chave pública e uma chave privada. Nesse modelo, o remetente utiliza a chave pública para cifrar os dados, enquanto o destinatário utiliza a chave privada correspondente para decifrá-los.

As duas chaves são matematicamente relacionadas, mas não são iguais, e possuem políticas distintas de confidencialidade: a chave privada deve ser mantida em sigilo, enquanto a chave pública pode ser compartilhada livremente.

A Figura 2.5 apresenta um exemplo de uso da criptografia assimétrica na transferência de informações entre duas partes (denominadas Alice e Bob), bem como a perspectiva de um terceiro não autorizado (denominado Charlie). Iniciando da esquerda para a direita, têm-se as seguintes ações:

- Alice deseja enviar a mensagem: “I will bid 1.2 million dollars”.
- Alice criptografa a mensagem utilizando a chave pública de Bob.
- A mensagem criptografada é gerada.
- Alice assina a mensagem com sua própria chave privada, garantindo a autenticidade.
- A mensagem criptografada e assinada é enviada para Bob.
- Bob verifica a assinatura digital utilizando a chave pública de Alice. Caso seja válida, Bob tem a garantia da autenticidade da mensagem.
- Bob descriptografa a mensagem utilizando sua chave privada.

Do ponto de vista de Charlie, mesmo que ele intercepte a mensagem criptografada, não conseguirá lê-la, pois não possui a chave privada de Bob. Tampouco conseguirá falsificar a assinatura que garante a autenticidade da mensagem, pois não possui a chave privada de Alice.

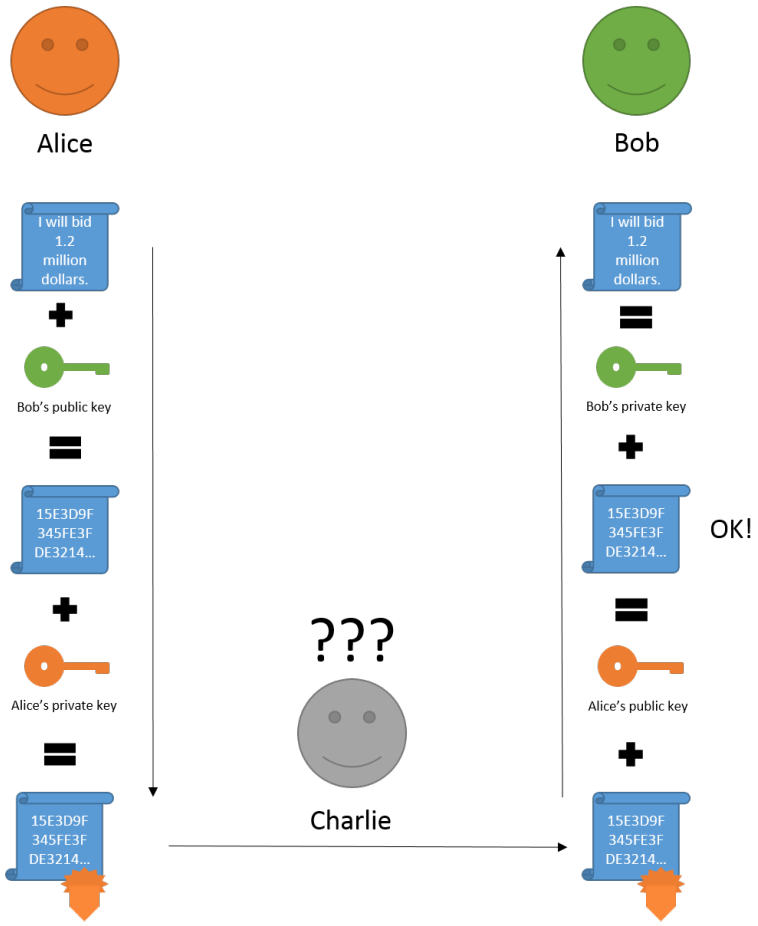

Figura 2.5 – Exemplo de um processo de encriptação Assimétrica (Fonte: [5]).

2.4.2 Criptografia Simétrica

Já na criptografia simétrica, é utilizada uma única chave para realizar tanto a criptografia quanto a descriptografia dos dados. Para que esse método funcione corretamente, todas as partes envolvidas na comunicação devem conhecer previamente essa chave. Para garantir a segurança da comunicação, essa chave não deve ser compartilhada com terceiros, sendo restrita somente aos participantes autorizados na transferência dos dados.

A Figura 2.6 apresenta um exemplo semelhante ao mencionado na seção anterior, mas agora utilizando criptografia simétrica. Iniciando da esquerda para a direita, tem-se que:

- Alice decide enviar a mensagem: "I will bid 1.2 million dollars". para Bob.
- Alice utiliza uma chave secreta (compartilhada) para criptografar a mensagem.
- A mensagem criptografada é gerada.
- Alice envia a mensagem criptografada para Bob.
- Charlie pode interceptar a mensagem, mas não pode lê-la, pois não possui a chave secreta.
- Bob, que possui a mesma chave secreta, utiliza-a para descriptografar a mensagem.

- Bob recupera o conteúdo original da mensagem: “I will bid 1.2 million dollar”.

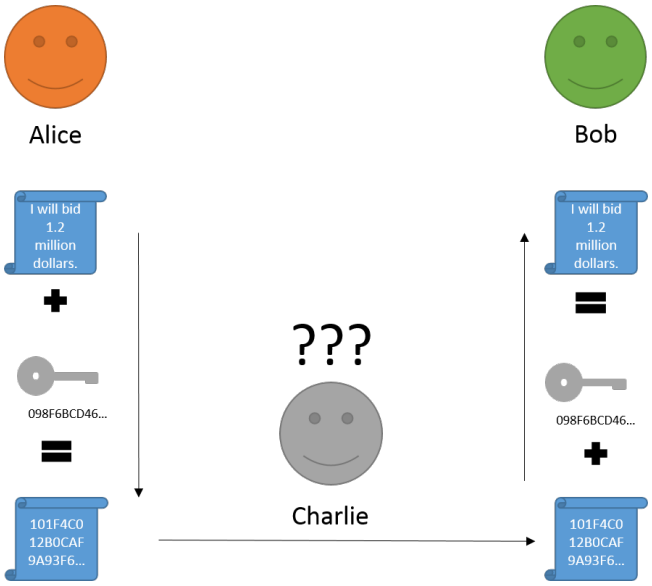

Figura 2.6 – Exemplo de um processo de encriptação Simétrica (Fonte: [5]).

O principal desafio desse método é que Alice e Bob precisam estabelecer uma forma segura de obter a chave compartilhada, sem que Charlie tenha acesso a ela. Caso Alice envie a chave por um canal inseguro e Charlie a intercepte, ele poderá não somente ler as mensagens trocadas entre Alice e Bob, mas também alterá-las.

Embora seja possível que Alice e Bob se reúnam previamente para definir uma chave compartilhada, essa abordagem é inviável na maioria dos cenários nos quais a criptografia simétrica é aplicada. Esse problema pode ser contornado com o uso de criptografia assimétrica, por meio de protocolos de troca de chaves, como o protocolo Diffie-Hellman. Nesse protocolo, as partes envolvidas (neste caso, Alice e Bob) estabelecem uma chave secreta compartilhada por meio da troca de valores públicos, sem que a chave em si seja transmitida diretamente.

Outro método possível é o uso do *RSA*, um sistema de criptografia de chave pública que também pode ser utilizado para proteger a troca de uma chave simétrica. Nesse caso, uma chave simétrica é gerada por uma das partes e criptografada com a chave pública do destinatário. A parte receptora utiliza sua chave privada para descriptografar essa chave simétrica. A partir desse ponto, a comunicação ocorre utilizando criptografia simétrica.

Para este TCC, escolheu-se como algoritmo de criptografia simétrica o *AES* (*Advanced Encryption Standard*), detalhado a seguir.

AES

Esta seção apresenta os conceitos gerais e o funcionamento do algoritmo *AES*. O algoritmo em si trabalha com blocos de dados fixos de 128 bits (16 bytes), organizados

internamente como uma matriz 4×4 de bytes, chamada de estado. Esse bloco de dados passa por uma série de transformações chamadas de rodadas, cujo número varia conforme o tamanho da chave utilizada, conforme apresentado na Tabela 2.1: 10 rodadas para chaves de 128 bits, 12 para chaves de 192 bits e 14 para chaves de 256 bits. Em cada rodada, o estado é transformado visando garantir confidencialidade e resistência a ataques criptográficos. A seguir, são explicados os passos do processo de encriptação.

Tabela 2.1 – Relação entre o tamanho da chave e o número de rodadas de encriptação no *AES*.

Tamanho da Chave (bits)	**Numero de Rodadas**
128	10
192	12
256	14

O processo de criptografia do *AES* consiste em várias operações fundamentais, conforme ilustrado na Figura 2.7.

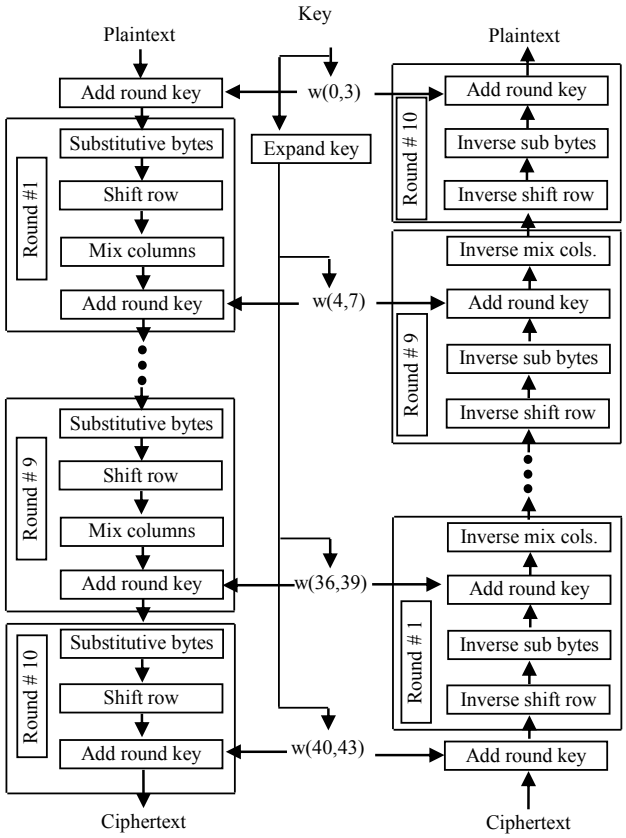

Figura 2.7 – Diagrama de blocos do processo de encriptação do algoritmo *AES* (Fonte: [1].)

A operação *AddRoundKey* gera uma subchave para a rodada atual, combinada com o estado por meio de uma operação *XOR* bit a bit.

A transformação *SubBytes*, também conhecida como S-box (*substitution box*), é um mapeamento bijetivo não linear que garante uma correspondência um-para-um entre

entrada e saída. Essa não linearidade é essencial para fornecer resistência contra ataques criptográficos. A geração da *S-box* é computacionalmente custosa, por envolver uma inversão no campo de Galois $GF(2^8)$ seguida de uma multiplicação de matrizes; por isso, a S-box é geralmente pré-calculada.

A operação *ShiftRows* desloca ciclicamente a segunda, terceira e quarta linhas da matriz de estado em 1, 2 e 3 bytes para a esquerda, respectivamente, contribuindo para a difusão. Já a operação *MixColumns* aplica uma multiplicação de matrizes em $GF(2^8)$ com constantes predefinidas sobre cada coluna do estado, promovendo maior propagação das alterações no bloco de dados.

2.5 Protocolo de Aplicação

Nesta seção, é explicado o protocolo *HTTP* e seu funcionamento, assim como duas versões específicas desse protocolo: *HTTP/2* e *HTTP/3*, relevantes para o entendimento do protocolo QUIC, abordado no próximo capítulo.

2.5.1 HTTP

O *HTTP* (*Hypertext Transfer Protocol*) é um protocolo criado em 1989 por Tim Berners-Lee para atuar na camada de Aplicação (7ª camada do modelo OSI / 4ª camada do modelo *TCP/IP*). A criação do protocolo foi fundamental para o desenvolvimento e funcionamento da Internet atual, pois seu objetivo inicial era permitir a busca de recursos como documentos *HTML*, além de viabilizar a troca de dados na Web. O protocolo opera no modelo cliente-servidor, como ilustrado na Figura 2.8, em que as solicitações são iniciadas pelo cliente, geralmente um navegador, e respondidas pelo servidor.

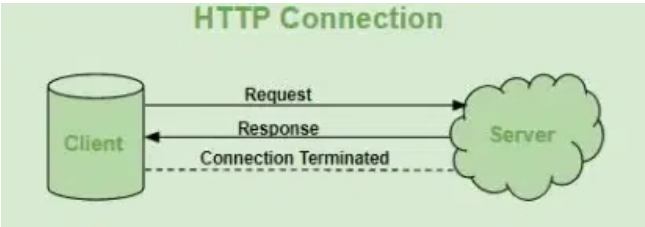

Figura 2.8 – Conexão *HTTP* (Fonte: [9]).

Seguindo esse modelo de operação, o protocolo se consolidou como a base da comunicação de dados na Internet, estabelecendo um padrão entre o cliente (navegador) e o servidor web para a comunicação e troca de dados. Desde sua criação, o protocolo passou por diversas atualizações para acompanhar a expansão da Internet, com várias versões publicadas ao longo dos anos, conforme listado abaixo:

- **HTTP/0.9**: Introduzida em 1991 como a versão inicial do protocolo.
- **HTTP/1.0**: Introduzida em 1996, com a publicação do *RFC* 1945 pela *IETF*.
- **HTTP/1.1**: Introduzida em 1997, por meio do *RFC* 2068, e posteriormente refinada no RFC 2616, em junho de 1999.
- **HTTP/2**: Introduzida em maio de 2015, com a publicação do *RFC* 7540.
- **HTTP/3**: Versão mais recente do protocolo, introduzida em junho de 2022, com a publicação do *RFC* 9114. Sua construção é baseada no protocolo QUIC, desenvolvido pela Google.

Duas das versões listadas acima (*HTTP/2* e *HTTP/3*) serão explicadas nos tópicos a seguir, por estarem diretamente relacionadas ao desenvolvimento do protocolo QUIC.

2.5.2 HTTP/2

O HTTP/2 foi desenvolvido em 2015 com o objetivo de solucionar problemas identificados na versão anterior do protocolo (*HTTP/1.1*). Seu foco principal foi manter a compatibilidade com aplicações Web existentes, ao mesmo tempo em que permitia que novas aplicações se beneficiassem dos recursos aprimorados introduzidos.

Isso foi possível porque as mudanças ocorreram no nível da estrutura de dados, ou seja, na forma como os dados são organizados e transportados entre cliente e servidor, sem alterar aspectos da codificação do protocolo, como métodos e campos de cabeçalho. As principais alterações estão listadas a seguir:

- **Priorização**: O *HTTP/2* introduziu a priorização ponderada, permitindo que os desenvolvedores definam a ordem de carregamento dos recursos de uma página com maior granularidade. Isso melhora tanto a velocidade percebida quanto a real de carregamento. O protocolo permite o envio simultâneo de múltiplos *streamings* de dados por meio de uma única conexão, e cada um deles pode receber um valor de prioridade, indicando ao cliente qual deve ser processado primeiro.
- **Multiplexação**: No *HTTP/1.1*, os recursos eram carregados sequencialmente, o que causava o problema conhecido como *head-of-line blocking*, na qual a falha ou lentidão na transmissão de um recurso impedia o carregamento dos seguintes. O *HTTP/2* resolveu esse problema no nível do protocolo *HTTP*, implementando a multiplexação, que permite o envio de vários *streamings* de dados simultaneamente em uma única conexão *TCP*. Cada *streaming* é numerado, permitindo que o cliente identifique corretamente as mensagens recebidas. No entanto, essa solução não elimina o problema de bloqueio no nível do *TCP*, pois a perda ou atraso de pacotes ainda afeta toda a conexão.
- **Push de servidor**: Diferentemente das versões anteriores, em que o servidor apenas respondia a solicitações, o *HTTP/2* introduziu a funcionalidade de *server push*, permi-

tindo que o servidor envie proativamente recursos ao cliente antes mesmo que sejam solicitados. Isso reduz a latência em páginas que requerem múltiplos recursos. O servidor também envia uma prévia (informação de cabeçalho) para indicar ao cliente qual conteúdo está sendo enviado.

- **Compactação de cabeçalhos**: Embora o *HTTP/1.1* já utilizasse formas de compactação, o *HTTP/2* implementou o método *HPACK*, que reduz significativamente a sobrecarga dos cabeçalhos *HTTP*. Esse método elimina redundâncias por meio de compactação, utilizando tabelas estáticas e dinâmicas, além de codificação de Huffman. A técnica visa reduzir o volume de dados transmitidos, especialmente útil em aplicações que realizam múltiplas requisições com cabeçalhos semelhantes.

Com essas mudanças, o *HTTP/2* tornou-se mais rápido e eficiente que sua versão anterior, com melhor desempenho na comunicação entre cliente e servidor. No entanto, com o passar do tempo, essa versão também passou a enfrentar limitações, especialmente devido ao uso de uma única conexão *TCP*, que ainda apresenta vulnerabilidades ao bloqueio de cabeça de fila em nível de transporte.

Dessa necessidade, surgiu o HTTP/3, a terceira e atual versão do protocolo *HTTP*, abordada no próximo tópico. Ele foi projetado para oferecer maior velocidade e segurança, superando limitações estruturais herdadas das versões anteriores.

2.5.3 HTTP/3

O *HTTP/3* é a versão mais recente do protocolo *HTTP*. Foi introduzido como um *Internet Draft* em 2022, sendo resultado direto da integração com o protocolo de transporte QUIC. Muitas das funcionalidades aprimoradas do *HTTP/3* derivam do uso do QUIC, cujos detalhes serão explorados no próximo capítulo.

A motivação para a criação de uma terceira versão do *HTTP* decorre dos resultados expressivos obtidos com o QUIC, especialmente em relação à velocidade, segurança e desempenho. As principais vantagens do *HTTP/3* em relação ao *HTTP/2* estão listadas a seguir:

- **Multiplexação em nível de transporte:** Diferentemente do *HTTP/2*, o *HTTP/3* foi projetado para eliminar o problema de *head-of-line blocking* no nível do transporte. Isso é possível porque, com o QUIC, cada *stream* (canal) é independente. Caso ocorra perda de pacotes em uma *stream* específica, essa perda não afeta as demais. O protocolo identifica qual fluxo foi afetado e realiza a retransmissão somente nesse canal, mantendo a independência dos fluxos paralelos, uma funcionalidade exclusiva do protocolo QUIC.
- **Handshake mais rápido e seguro:** O QUIC incorpora o protocolo *TLS 1.3* diretamente em sua estrutura, ao contrário do *HTTP/2*, em que o *TLS* é utilizado como uma

camada adicional. Isso permite conexões mais rápidas e seguras, com suporte ao modo *0-RTT* (*zero round-trip time*), que reduz significativamente a latência na inicialização de conexões seguras.

- **Migração de conexão:** O QUIC permite a continuidade das conexões mesmo em casos de mudança de endereço *IP*, por meio do uso de *Connection IDs*, que são identificadores persistentes da conexão. Isso elimina a necessidade de um novo *handshake*, promovendo maior estabilidade em redes móveis e ambientes com troca frequente de endereçamento.

A Figura 2.9 apresenta uma comparação do encapsulamento nas três versões do protocolo HTTP discutidas nesta seção.

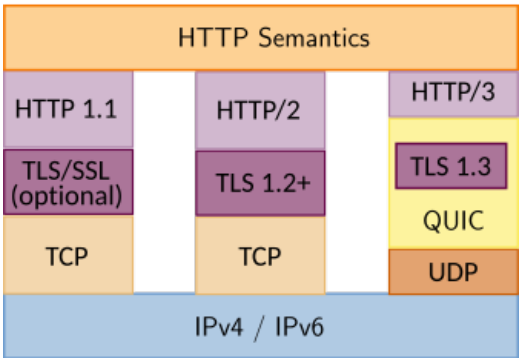

Figura 2.9 – Comparação das versões *HTTP/1.1*, *HTTP/2* e *HTTP/3* (Fonte: [6]).

Observa-se uma diferença significativa entre as versões no que diz respeito à usabilidade e à estrutura do protocolo. Nas versões *HTTP/1.1* e *HTTP/2*, os componentes atuam modularmente, com cada protocolo ocupando uma camada específica da pilha de rede e interagindo de maneira relativamente independente. Essa organização favorece a manutenção e o diagnóstico, mas também impõe limitações em termos de desempenho e flexibilidade.

No HTTP/3, essa separação é menos rígida: o protocolo QUIC é executado sobre o *UDP*, substituindo o *TCP* como camada de transporte, mas incorporando funcionalidades típicas do *TCP*, como confiabilidade na entrega, controle de congestionamento e ordenação de pacotes. Essas características são implementadas diretamente no nível da aplicação, o que permite maior controle sobre o comportamento da conexão.

Além disso, o *TLS* 1.3 deixa de ser uma camada separada e compõe a estrutura interna do protocolo QUIC, sendo parte essencial de seu processo de estabelecimento de conexão. Essa integração resulta em negociações de segurança mais rápidas, menor latência e maior resistência a ataques de *downgrade*, contribuindo para um modelo de comunicação mais eficiente e seguro.

O próximo capítulo é dedicado ao protocolo QUIC, abordando seu funcionamento, estrutura de datagrama, vantagens e desvantagens em relação aos protocolos anteriores.

3. PROTOCOLO QUIC

Este capítulo apresenta o protocolo utilizado neste Trabalho de Conclusão de Curso: o “Quick UDP Internet Connection”, mais conhecido como QUIC.

3.1 Introdução

O protocolo QUIC iniciou em 2012, desenvolvido pela Google como uma solução para compensar deficiências observadas no uso do *HTTP/2*. O objetivo principal era oferecer uma comunicação mais eficiente, com encriptação robusta e redução da perda de pacotes durante os *handshakes* entre cliente e servidor nos serviços web da empresa. O protocolo permaneceu de uso exclusivo da Google até maio de 2021, quando foi oficialmente padronizado pela IETF (*Internet Engineering Task Force*) como sucessor do *HTTP/2*, compondo a base do *HTTP/3*. Sua especificação foi publicada sob o *RFC* 9000 [14].

Conforme a definição da *IETF*, o QUIC é um protocolo orientado à conexão criptografada, operando na quarta camada do modelo OSI (camada de transporte). A principal motivação para sua criação foi a redução da latência, fator cada vez mais crítico para serviços que dependem de comunicação em tempo real ou carregamento rápido de conteúdos via Internet.

O protocolo foi concebido para substituir o modelo tradicional baseado em *TCP + TLS*, amplamente utilizado nas comunicações via *HTTP/2*. Embora esse modelo ofereça segurança e confiabilidade, ele é penalizado por sua alta latência, já que depende da execução do *three-way handshake* do *TCP*, seguida da negociação de parâmetros criptográficos via *TLS* — como descrito na Seção 2.3.

A estratégia adotada para reduzir a latência foi implementar o QUIC sobre o protocolo de transporte *UDP*, conforme discutido na Seção 2.1.1. O *UDP* não requer *handshakes* complexos para estabelecer conexões, contribuindo para inicializações mais rápidas. No entanto, sua utilização traz desafios relacionados à segurança e à confiabilidade da comunicação, pois, em sua configuração padrão, o *UDP* não garante a entrega ordenada e segura dos pacotes.

Para superar essa limitação, o *TLS* foi incorporado diretamente ao QUIC, sendo executado no processo de *handshake* inicial do protocolo. Dessa forma, a troca de chaves e a negociação de segurança ocorrem já no momento de estabelecimento da conexão, sem a necessidade de múltiplas etapas externas.

Outro desafio enfrentado foi a ausência, no *UDP*, de mecanismos nativos de controle de perda de pacotes. Para contornar essa limitação, o QUIC implementa internamente

funcionalidades de detecção e retransmissão de pacotes perdidos, assegurando maior confiabilidade na comunicação.

3.2 QUIC versus TCP+TLS

Esta seção compara a comunicação que utiliza o protocolo QUIC e outra baseada no modelo *TCP + TLS*. O objetivo é explicar o funcionamento das etapas do *handshake* do QUIC e compará-las com o modelo tradicional utilizado no *TCP + TLS*, conforme abordado na Seção 2.3.

Com base na Figura 3.1, observa-se que o *handshake* do protocolo QUIC é significativamente mais rápido. Isso se deve, conforme discutido na Figura 2.9, ao fato de o QUIC operar sobre o protocolo *UDP*. Assim, ele não depende das etapas de validação exigidas pelo *TCP* para estabelecer uma conexão, como o *three-way handshake*.

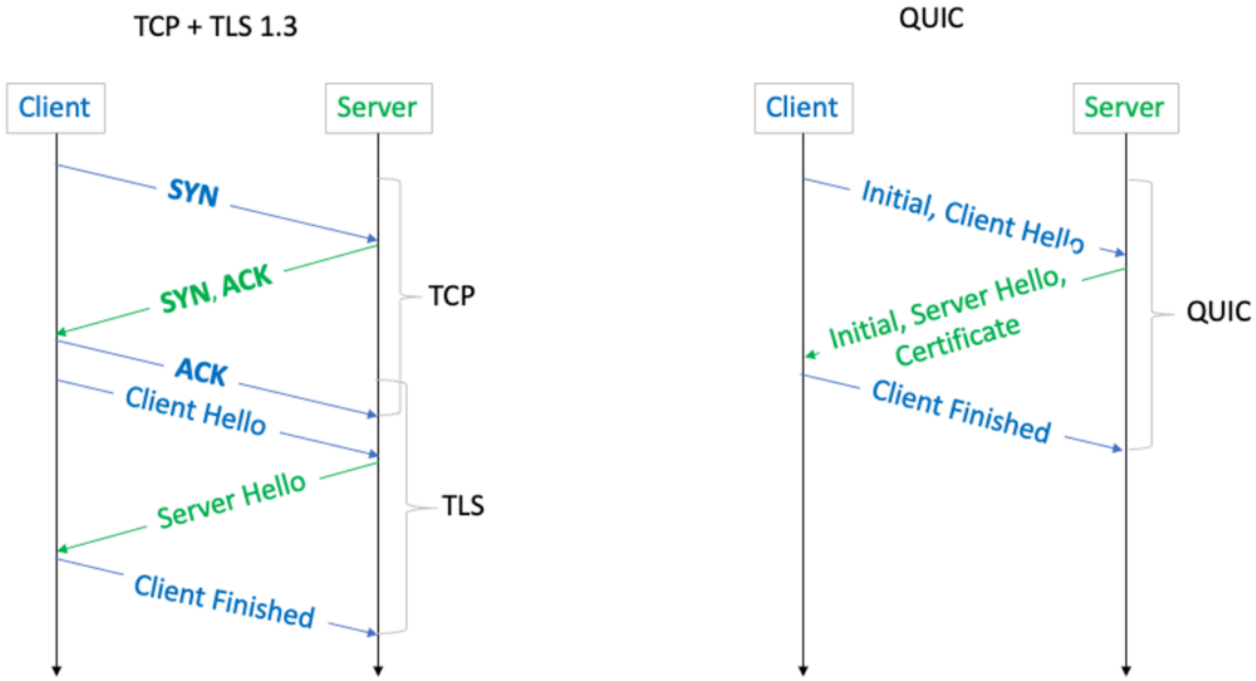

Figura 3.1 – Processos de *handshake*: QUIC x TCP + TLS (Fonte: [10]).

Além disso, como mencionado na introdução deste capítulo, o protocolo *TLS* (Seção 2.3) foi incorporado diretamente ao QUIC. Isso elimina a necessidade de uma etapa adicional de negociação de segurança após o estabelecimento da conexão, como ocorre no modelo *TCP + TLS*. No QUIC a autenticação e a configuração da segurança da conexão ocorrem integradamente, durante o próprio *handshake*, sem a necessidade de mensagens extras entre cliente e servidor.

A seguir, com base na imagem anterior, detalham-se as etapas do *handshake* do protocolo QUIC:

- **1. Envio de "Initial" e "Client Hell"**:
  - O cliente inicia a conexão enviando um pacote QUIC do tipo *Initial*.
  - Este pacote já contém a mensagem *Client Hello*, primeira etapa do *handshake* do *TLS* 1.3.
  - A mensagem inclui:
    - Versão do protocolo *TLS* suportada;
    - Algoritmos de criptografia suportados (ex: *AES-128*);
    - Parâmetros para a geração de chave, com base em Diffie-Hellman;
    - Identificador da conexão QUIC (*Connection ID*).
- **2. Resposta com "Initial", "Server Hello" e "Certificate"**:
  - O servidor responde com um pacote também do tipo *Initial*.
  - Este pacote contém:
    - A mensagem *Server Hello* do *TLS 1.3*, com a escolha do algoritmo de criptografia;
    - O certificado digital do servidor, necessário para autenticação;
    - Outras mensagens *TLS*, como *Certificate Verify*, que comprova a posse da chave privada correspondente ao certificado.
  - O cliente, ao receber essa resposta, valida a identidade do servidor e deriva as chaves para comunicação criptografada.
- **3. Envio de "Client Finished"**:
  - Após a validação do certificado e a geração das chaves simétricas, o cliente envia a mensagem *Client Finished*.
  - Essa mensagem indica o término do *handshake TLS*.
  - A partir deste ponto, cliente e servidor compartilham dados criptográficos e podem iniciar a troca de dados protegidos.

Essas são as etapas realizadas pelo protocolo QUIC para estabelecer uma conexão segura entre cliente e servidor.

3.3 Datagrama QUIC

Esta seção detalha a estrutura do pacote QUIC encapsulado em um datagrama *UDP*. O objetivo é apresentar como o protocolo realiza as etapas de *handshake* descritas na seção anterior. Utilizando a Figura 3.2 como referência, a explicação inicia pelo conteúdo do datagrama *UDP* e segue até os componentes de cada *frame* de um pacote QUIC.

- **Datagrama UDP:** Unidade de transmissão do protocolo *UDP* que encapsula os dados do QUIC.

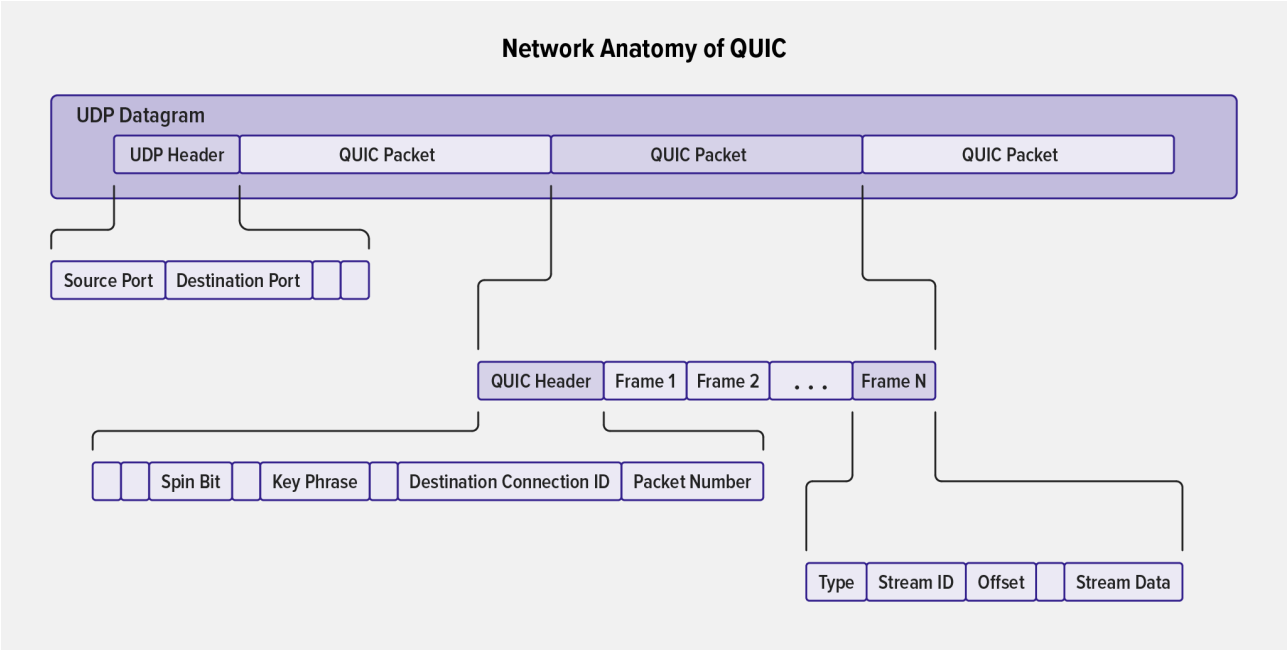

Figura 3.2 – Composição do protocolo QUIC utilizando um *short header* (Fonte: [16]).

  - **UDP Header:** Cabeçalho do protocolo *UDP* contendo informações essenciais para roteamento e entrega.
    - **Source Port:** Porta de origem do pacote.
    - **Destination Port:** Porta de destino do pacote.
    - **Length:** Especifica o comprimento total do datagrama *UDP* (16 bits), incluindo o cabeçalho e os dados. Permite ao receptor delimitar o final do datagrama.
    - **Checksum:** Campo de 16 bits utilizado como mecanismo de detecção de erros, verificando a integridade do cabeçalho e dos dados.
  - **QUIC Packet:** Pacote do protocolo QUIC encapsulado no campo de dados do datagrama *UDP*. Um datagrama pode conter um ou mais pacotes QUIC, compostos por um cabeçalho e um ou mais *frames*.

O **QUIC Packet** contém os seguintes campos:

- **QUIC Header:** Cabeçalho do pacote QUIC, com duas variações utilizadas em diferentes fases da conexão: **Long Header** e **Short Header**.
  - **Long Header:** Utilizado durante o estabelecimento da conexão (ex.: *Initial*, *0-RTT*, *Handshake*, *Retry*).
    - **Header Form (1 bit):** Define o uso de *Long Header* (valor 1).
    - **Fixed Bit (1 bit):** Sempre definido como 1.
    - **Long Packet Type (2 bits):** Indica o tipo de pacote.
    - **Reserved Bits (2 bits):** Reservado para uso futuro.
    - **Packet Number Length (2 bits):** Tamanho do campo de número de pacote.

        - **Version (32 bits):** Versão do protocolo QUIC.
        - **Destination Connection ID Length (8 bits):** Tamanho do identificador de destino.
        - **Destination Connection ID (0–160 bits):** Identificador da conexão de destino.
        - **Source Connection ID Length (8 bits):** Tamanho do identificador de origem.
        - **Source Connection ID (0–160 bits):** Identificador da conexão de origem.
        - **Token Length (variável):** Tamanho do *token* (apenas em *Initial*).
        - **Token (variável):** *Token* usado para reidentificação.
        - **Length (variável):** Comprimento do restante do pacote (*payload*).
        - **Packet Number (8–32 bits):** Número sequencial do pacote.
    - **Short Header:** Utilizado após o *handshake*.
        - **Header Form (1 bit):** Indica *Short Header* (valor 0).
        - **Fixed Bit (1 bit):** Sempre igual a 1.
        - **Spin Bit (1 bit):** Utilizado para medições de latência sem estado.
        - **Reserved Bits (2 bits):** Reservado para extensões futuras.
        - **Key Phase (1 bit):** Indica a fase da chave criptográfica em uso.
        - **Packet Number Length (2 bits):** Tamanho do campo de número de pacote.
        - **Destination Connection ID (0–160 bits):** Identificador da conexão no destinatário.
        - **Packet Number (8–32 bits):** Número do pacote, utilizado para controle de fluxo e retransmissão.
- **Frames:** Estrutura de dados que transporta informações da aplicação ou de controle no pacote QUIC.
    - **Frame 1, Frame 2, ..., Frame N:** Múltiplos *frames* podem estar presentes em um único pacote.
    - **Cada frame pode conter:**
        - **Type (8 bits):** Indica o tipo de *frame* (ex.: `STREAM`, `ACK`, `PING`).
        - **Stream ID (variável):** Identificador do fluxo ao qual o *frame* pertence.
        - **Offset (variável):** Posição dos dados dentro do fluxo (presente em `STREAM` frames).
        - **Length (variável):** Tamanho dos dados transportados.
        - **Stream Data (variável):** Dados efetivos transmitidos.

Resumidamente, o protocolo QUIC opera na área de dados do datagrama *UDP*, podendo conter pacotes com dois tipos distintos de cabeçalho: o *long header* e o *short header*. O *long header* é utilizado durante a fase de estabelecimento da conexão, sendo empregado nos pacotes dos tipos `Initial`, `0-RTT`, `Handshake` e `Retry`. Esses pacotes desempenham funções específicas no processo de autenticação, compartilhando campos em comum, embora suas estruturas variem conforme o tipo de pacote.

Uma vez concluído o *handshake*, os pacotes subsequentes passam a utilizar o *short header*, uma versão mais compacta, aplicada exclusivamente nos pacotes do tipo `1-RTT`, utilizados na transmissão de dados já protegidos por criptografia.

A seguir, serão apresentados e detalhados os pacotes `Initial`, `0-RTT`, `Handshake`, `Retry` e `1-RTT`, com suas respectivas funções no processo de conexão segura do protocolo QUIC.

3.4 Pacote do Protocolo QUIC

Esta seção apresenta os pacotes `Initial`, `Retry`, `0-RTT`, `Handshake`, e `1-RTT` do protocolo.

3.4.1 Initial

A função do protocolo *Initial* é iniciar a conexão QUIC e transportar a primeira mensagem do *handshake* (TLS ClientHello) e também negociar a versão do protocolo. Os campos que compõem esse pacote são:

- **Initial Header:**
  - Header Form.
  - Fixed Bit.
  - Long Packet Type = Valor 0b00 para identificar um pacote *Initial*.
  - Reserved Bits.
  - Packet Number Length.
  - Version.
  - Destination Connection ID Length.
  - Destination Connection ID.
  - Source Connection ID Length.
  - Source Connection ID.
  - **Token Length (1 a 8 bytes):** Um inteiro de tamanho variável que especifica o comprimento do campo *Token*, em bytes. Esse valor é 0 se nenhum *token* estiver sendo utilizado. Pacotes do tipo *Initial* enviados pelo servidor devem definir o campo *Token Length* como 0; clientes que receberem um pacote *Initial* com um campo *Token Length* diferente de zero devem descartá-lo.
  - **Token (0 a 65535 bytes):** Campo que transporta o *token* de validação (em *Initial*) ou o *Retry Token* (em *Retry*).
  - Length.
  - Packet Number.

  - **Payload:** Transporta as primeiras mensagens do protocolo *TLS 1.3* para iniciar o processo de *handshake*("Hello Server").

Daremos ênfase a dois campos citados anteriormente: *Token Length* e *Token*, pois ambos funcionam como mecanismos de reconexão. Quando utilizados, esses campos permitem ao servidor verificar se o cliente ainda possui o mesmo endereço *IP* utilizado em uma tentativa anterior de conexão, ou em uma conexão previamente estabelecida.

3.4.2 Retry

O pacote *Retry* visa, em um contexto cliente-servidor, validar a autenticidade da tentativa de conexão. Esse processo ocorre quando o cliente envia um pacote do tipo *Initial* com carga útil, porém sem o campo *Token* preenchido. Ao receber esse pacote, o servidor verifica se o cliente possui realmente o endereço *IP* declarado, como forma de proteção contra ataques de *IP spoofing* — técnica em que um atacante falsifica o endereço *IP* de origem para se passar por outro dispositivo. Nesse caso, o servidor responde com um pacote *Retry*, que contém um *Token* gerado por ele. Ao receber esse pacote, o cliente deve reenviar um novo pacote *Initial*, agora com o campo *Token* preenchido com o valor recebido anteriormente no pacote *Retry*. É importante destacar que o servidor não prossegue com o estabelecimento da conexão até que o cliente envie um pacote *Initial* válido contendo o *Token* correto. Além disso, como o pacote *Retry* não transporta dados de aplicação e não faz parte da contagem de pacotes com criptografia *TLS*, o seu número de pacote é independente e não é criptografado. O cliente, ao reenviar o novo pacote *Initial*, redefine o número do pacote para zero, reiniciando a contagem, uma vez que o fluxo criptográfico anterior foi interrompido e reiniciado após o *Retry*. A seguir, apresenta-se a estrutura do cabeçalho desse tipo de pacote:

- **Retry Header:**
  - Header Form.
  - Fixed Bit.
  - Long Packet Type: Valor 0b11 para identificar um pacote *Retry*.
  - Version.
  - Destination Connection ID Length.
  - Destination Connection ID.
  - Source Connection ID Length.
  - Source Connection ID.
  - **Original Destination Connection ID (0 a 20 bytes)**: Garante integridade de quem está tentando conectar, esse campo contém o *Destination Connection ID* que o cliente usou originalmente no pacote *Initial*.

  - **Retry Token (Mesmo tamanho do campo token, visto no pacote Initial)**: *Token* criptograficamente protegido. É um tipo específico de *Token*, enviado somente no pacote *Retry*.
  - **Retry Integrity Tag (16 bytes)**: *Tag* de integridade para proteger contra falsificações. Essa *tag* é gerada a partir de um calculo (utilizando o algoritmo de autenticação *AEAD*, no qual será comentado no próximo tópico), realizado pelo servidor para proteger a integridade do pacote *Retry* contra modificações.
  - **Payload:** Um *Retry Token* que o cliente deve incluir em um novo *Initial*.

Os campos relevantes nesse pacote são os 3 últimos, como visto na lista acima: Original *Destination Connection ID*, *Retry Token* e *Retry Integrity Tag*. Esses campos são os que garantem a autenticidade e a integridade da resposta do servidor antes da conexão com o cliente ser estabelecida.

3.4.3 0-RTT

O *zero round-trip time*, ou simplesmente *0-RTT*, tem por função, no processo de estabelecimento de conexão, permitir que o cliente envie dados antecipadamente, antes da conclusão do *handshake*, reutilizando informações de uma conexão anterior. Em outras palavras, caso o cliente já tenha se conectado anteriormente ao servidor, ele poderá utilizar os dados dessa sessão para retomar a comunicação sem a necessidade de repetir todas as etapas do *handshake*.

Caso o cliente receba um pacote *Retry* do servidor, os pacotes *0-RTT* previamente enviados são considerados perdidos ou descartados. Nessa situação, o cliente deve reenviar os dados por meio de novos pacotes *0-RTT*, após o envio de um novo pacote *Initial*. É importante destacar que esses reenvios devem utilizar novos números de pacote, pois a reutilização pode comprometer a segurança da criptografia aplicada aos dados.

A estrutura do pacote *0-RTT*, em termos de campos, segue o mesmo formato do cabeçalho do tipo *Long Header*, não havendo diferenças estruturais em relação aos demais pacotes desse tipo.

- **0-RTT Header:**
  - Header Form
  - Fixed Bit
  - Long Packet Type = Valor 0x1 para identificar um pacote *0-RTT*.
  - Reserved Bits
  - Packet Number Length
  - Version
  - Destination Connection ID Length
  - Destination Connection ID

  - Source Connection ID Length
  - Source Connection ID
  - Length
  - Packet Number
  - **Payload:** Dados que o cliente deseja que servidor receba antes de finalizar o handshake(Ex: uma requisição HTTP).

3.4.4 Handshake

O último pacote específico que utiliza o cabeçalho do tipo *long* é o pacote *Handshake*. Esse pacote é utilizado durante o processo de estabelecimento seguro da conexão, após o envio bem-sucedido do pacote *Initial*. Ele contém mensagens do protocolo *TLS 1.3*, criptografadas com chaves derivadas do segredo inicial trocado entre cliente e servidor. Essas mensagens são essenciais para completar a autenticação das partes e finalizar a negociação das chaves criptográficas. A recepção de um pacote *Handshake* permite que cliente e servidor avancem no processo de criptografia da conexão, substituindo as chaves derivadas do segredo inicial (usadas em *0-RTT*) por chaves mais seguras, chamadas de *1-RTT*, que são obtidas após a realização completa do processo de derivação de chaves, incluindo autenticação mútua e *forward secrecy*. A *1-RTT key* é mais segura porque é derivada após a confirmação da identidade do servidor, e utiliza chaves temporárias *ECDHE* — que garantem sigilo mesmo se as chaves de longo prazo forem comprometidas futuramente. A partir desse ponto, os pacotes passam a ser protegidos com cabeçalhos do tipo short (*Short Header*), utilizando os pacotes *1-RTT* para a troca de dados.

- **Handshake Packet:** Utilizado para transportar mensagens do *TLS 1.3* durante o processo de handshake.
  - Header Form
  - Fixed Bit
  - Long Packet Type: Especifica o tipo do pacote — neste caso, *Handshake* (valor 0x02).
  - Reserved Bits
  - Packet Number Length
  - Version
  - Destination Connection ID Length
  - Destination Connection ID
  - Source Connection ID Length
  - Source Connection ID
  - Length
  - Packet Number

  - **Payload (TLS Handshake data):** Contém os frames criptografados com dados de handshake do TLS ("Server Hello"ou "Finished").

3.4.5 1-RTT

Por fim, temos o pacote *1-RTT*, que é o único tipo de pacote utilizado com o cabeçalho do tipo Short Header. Esse pacote se torna o principal meio de transporte de dados em uma conexão QUIC após o estabelecimento seguro da conexão. Mais especificamente, após a conclusão bem-sucedida da troca de mensagens TLS — por meio dos pacotes *Initial* e *Handshake* — o cliente e o servidor criam novas chaves de criptografia baseando-se nas chaves antigas(*0-RTT*), chamadas de chaves *1-RTT*, com o intuito de substitui-las . A partir de então, os pacotes são criptografados e autenticados com as chaves *1-RTT*, garantindo a confidencialidade, integridade e autenticidade dos dados transmitidos.

- **1-RTT Packet (Short Header):** Utilizado após a conclusão do *handshake*, para transmitir dados protegidos com as chaves finais (*1-RTT*).
  - **Header Form (1 bit):** Sempre 0, indicando que é um pacote do tipo *Short Header*.
  - Fixed Bit
  - **Spin Bit (1 bit):** Opcional, utilizado para medir *RTT* passivamente na rede.
  - Reserved Bits
  - **Key Phase (1 bit):** Indica qual conjunto de chaves está sendo usado (importante para alternância de chaves).
  - Packet Number Length
  - Destination Connection ID
  - Packet Number
  - **Packet Payload (8 bits ou mais):** Dados criptografados transportando *frames*, como *STREAM*, *ACK*, *PING*, etc.

3.5 Frames

Os *frames* são as unidades fundamentais que compõem o *payload* dos pacotes utilizados pelo protocolo QUIC. Cada *frame* serve a um propósito específico, como carregar dados da aplicação, gerenciar o fluxo de dados, realizar controle de congestionamento ou manter o estado da conexão. Os *frames* são independentes e podem ser empacotados sequencialmente dentro de um mesmo pacote QUIC. Ao contrário do *TCP*, que possui um único fluxo de dados por conexão, o QUIC utiliza múltiplos *streams* multiplexados, cada um representado por *frames* do tipo *STREAM*, permitindo a transmissão paralela de dados. Dessa forma, se ocorrer uma perda de pacote em um *stream*, os demais não são afetados,

o que evita o problema conhecido como *head-of-line blocking*. A seguir, veremos alguns dos principais tipos de *frames* utilizados pelo protocolo.

- ***Stream* (0x08 - 0x0f):** Transporta dados da aplicação associados a um fluxo. Pode conter:
  - ***Stream ID*:** Identificador do fluxo ao qual os dados pertencem. Cada *stream* tem um ID único, que também indica a direção (unidirecional ou bidirecional) e a origem (cliente ou servidor) com base em seus bits menos significativos.
  - ***Offset*:** Indica a posição do primeiro byte de dados transportado por este *frame* dentro do fluxo. Ele permite que os dados sejam remontados corretamente no destino, mesmo que cheguem fora de ordem. Por exemplo, se o *offset* for 0, significa que os dados começam no início do *stream*.
  - ***Length*:** (Opcional) Especifica o número de bytes de dados contidos neste *frame*. Esse campo pode ser omitido se o *frame* se estende até o final do pacote. Serve para delimitar o tamanho exato do campo de dados.
  - ***Data*:** Contém os dados da aplicação que estão sendo transmitidos. Essa é a parte efetivamente útil do *frame*, transportando as informações do cliente para o servidor (ou vice-versa).
  - ***FIN bit*** (se for o fim do fluxo): (Opcional) Um bit que, quando definido, indica que este é o último *frame* deste *stream*. Ele informa ao receptor que não há mais dados a serem esperados nesse fluxo específico, encerrando o *stream*.
- ***ACK* (0x02, 0x03):** Informa quais pacotes foram recebidos, permitindo o controle de confiabilidade.
- ***CRYPTO* (0x06):** Transporta mensagens criptográficas TLS (como *ClientHello*, *ServerHello*, *Finished*). Similar ao *STREAM*, mas sem *Stream ID*.
- ***PADDING* (0x00):** Adiciona preenchimento ao pacote. Usado para alinhar tamanhos ou mascarar comprimento real.
- ***PING* (0x01):** Mantém a conexão ativa ou testa a latência, sem carregar dados.
- ***CONNECTION CLOSE* (0x1c, 0x1d):** Informa encerramento da conexão. Pode ser por erro de transporte ou da aplicação.
- ***NEW CONNECTION ID* (0x18):** Informa novos *Connection IDs* válidos que podem ser usados no futuro.
- ***RETIRE CONNECTION ID* (0x19):** Indica que um *Connection ID* não deve mais ser usado.
- ***MAX STREAM DATA* (0x11):** Informa ao *peer* que ele pode enviar mais dados em um fluxo específico.
- ***MAX DATA* (0x10):** Aumenta o limite total de dados permitidos na conexão.
- ***STREAMS BLOCKED* (0x16, 0x17):** Indica que não é possível abrir novos fluxos por limites de fluxo.

- ***NEW TOKEN* (0x07):** Envia um *token* criptografado para que o cliente reutilize em futuras conexões.
- ***PATH CHALLENGE* (0x1a) e *PATH RESPONSE* (0x1b):** Testam a conectividade entre dois *endpoints* por novos caminhos.
- ***DATA BLOCKED* (0x14):** Indica que a conexão está bloqueada por limite de dados.
- ***MAX STREAMS* (0x12, 0x13):** Aumenta o número máximo de fluxos uni ou bidirecionais.
- ***RESET STREAM* (0x04):** Interrompe um fluxo prematuramente e informa ao *peer*.
- ***STOP SENDING* (0x05):** Solicita que o *peer* pare de enviar dados em um fluxo.

3.6 Encriptação dos pacotes QUIC

Após compreendermos a estrutura do protocolo, os diferentes tipos de pacotes e sua composição, é necessário entender como funciona a criptografia dos pacotes. Especificamente, quais campos desses pacotes são criptografados e quais algoritmos de criptografia são utilizados.

A criptografia é aplicada em praticamente todos os dados transportados, desde os primeiros momentos da conexão. Diferentemente do *TCP*, no qual a criptografia via *TLS* ocorre acima da camada de transporte, o QUIC incorpora o protocolo *TLS 1.3* diretamente em sua arquitetura. Isso permite que partes sensíveis do pacote (como o número do pacote, *frames*, *ACKs* e dados em geral) sejam criptografadas.

Essa criptografia é realizada por meio de algoritmos *AEAD (Authenticated Encryption with Associated Data)*, como *AES-128-GCM* ou *ChaCha20-Poly1305*. Cada tipo de pacote (*Initial*, *Handshake*, *0-RTT*, *1-RTT*) utiliza chaves distintas, derivadas de segredos trocados ou gerados durante o processo *TLS*.

Como resultado, o QUIC proporciona confidencialidade, integridade e autenticação dos dados transportados desde o início da comunicação, dificultando interceptações, injeções ou alterações maliciosas de pacotes.

A seguir, são apresentados os campos de cada tipo de pacote que são criptografados e os que permanecem em texto claro.

- ***Initial Packet***
  - **Criptografado:** *Payload* (*frames CRYPTO* com mensagens TLS)
  - **Criptografado:** Número do pacote (*Packet Number*)
  - **Não criptografado:** *Connection ID* (necessário para roteamento)
  - **Não criptografado:** Parte do *header*
  - **Chave usada:** *Initial Secret*, derivada do *Destination Connection ID (DCID)*

- ***Handshake Packet***
  - **Criptografado:** *Payload* (*frames CRYPTO* com *ServerHello*, *Finished*, etc.)
  - **Criptografado:** Número do pacote (*Packet Number*)
  - **Criptografado:** *Frames ACK*, se presentes
  - **Chave usada:** Chave derivada após a troca de *ServerHello*
- ***0-RTT Packet***
  - **Criptografado:** *Payload* (*frames STREAM* com dados da aplicação)
  - **Criptografado:** Número do pacote
  - **Atenção:** Menor segurança; suscetível a ataques de repetição (*replay*)
  - **Chave usada:** *0-RTT Secret*, derivada de uma conexão anterior
- ***1-RTT Packet (Short Header)***
  - **Criptografado:** *Payload* (*frames STREAM*, *ACK*, *PING*, etc.)
  - **Criptografado:** Número do pacote (ofuscado)
  - **Criptografado:** Todas as *frames* transportadas
  - **Chave usada:** Chave final de aplicação (mais segura)

O único pacote que não possui campos criptografados é o pacote *Retry*, uma vez que ele não carrega *payload* e sua função principal é auxiliar na validação do endereço IP do cliente, como medida de proteção contra ataques de IP *spoofing*. Portanto, não há necessidade de aplicar criptografia nesse pacote.

Na figura Figura 3.3, é apresentada uma comparação dos campos criptografados no protocolo *TCP* + *TLS* e no protocolo QUIC. Observa-se que, enquanto no *TCP* + *TLS* a criptografia ocorre apenas em nível de aplicação, restringindo-se ao *payload*, no QUIC a criptografia abrange tanto a camada de aplicação quanto parte da camada de transporte, assegurando maior confidencialidade e integridade dos dados.

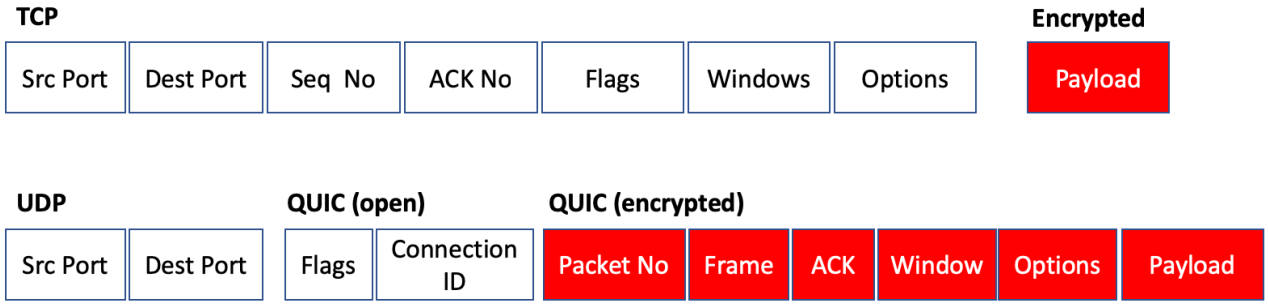

Figura 3.3 – Comparação da Encriptação dos protocolos TCP e QUIC (Source:[11]).

Por fim, a seguir será apresentado um exemplo de conexão utilizando o protocolo QUIC, onde o foco será na realização das etapas de criptografia citadas anteriormente.

- **1. Cliente inicia a conexão:**

  - O cliente deseja acessar `https://example.com`.
  - Ele envia um pacote *Initial* contendo a mensagem TLS *ClientHello*.
  - Não há uma proteção a esses dados ainda, então o *payload* é criptografado com *Initial keys* derivadas de um *salt* fixo + *Destination Connection ID*.
- **2. Construção do segredo inicial:**
  - O cliente utiliza um valor pré-definido (*salt*), definido pelo RFC do protocolo QUIC:

```
initial_salt = 0x38762cf7f55934b34d179ae6a4c80cadccbb7f0a
```

  - Ele combina esse valor com o valor armazenado no campo *Destination Connection ID* e aplica *HKDF (HMAC-based Key Derivation Function)*, que é uma função de derivação de chave baseada no algoritmo de autenticação *HMAC (Hash-based Message Authentication Code)*:

```
initial_secret = HKDF-Extract(initial_salt, dcid)
```

  - Com a derivação realizada, obtemos:
    * A chave de encriptação
    * *Initialization Vector* (conjunto de caracteres randômicos, normalmente usados para garantir que a criptografia sempre seja única)
    * Chave de proteção de *header*
- **3. Servidor responde:**
  - O servidor recebe o *Initial* e responde com outro pacote *Initial* (com *ServerHello*).
  - Ambos derivam chaves *handshake* com os parâmetros TLS.
- **4. Criptografia do *payload*:**
  - O *payload* pode conter um *frame CRYPTO* com o *handshake* TLS.
  - Esse *payload* é criptografado com *AES-128-GCM*:

```
ciphertext = AES-GCM(key, iv, plaintext, associated_data)
```

  - *associated data* = o *header* QUIC (não criptografado, mas autenticado).
- **5. Proteção do cabeçalho:**
  - O campo *Packet Number* (ex: 0x02) é criptografado usando a *header protection key*.
  - Os bits do *header* (como os *flags*) também são ofuscados.
- **6. Envio de pacotes *1-RTT*:**
  - Após o *handshake*, cliente e servidor derivam as *1-RTT keys*.
  - Pacotes de dados reais (como requisições HTTP) são criptografados com essas novas chaves.
  - Exemplo:
  - É colocado em um *STREAM Frame* e criptografado com as *1-RTT keys*.

4. IMPLEMENTAÇÃO

Com os conceitos do protocolo QUIC compreendidos, foi discutida, durante a execução do projeto SIGMARR, a possibilidade de implementação desse protocolo com o objetivo de utilizá-lo para resolver as questões de segurança do aplicativo, especificamente a criptografia dos dados transmitidos durante a comunicação entre o roteador e a aplicação.

Em sua configuração *default*, a aplicação já utilizava o programa *chownat* (Seção 2.2) para realizar a técnica de *hole punching* e permitir que o roteador efetuasse a troca de dados caso a aplicação estivesse em uma rede diferente da do roteador. No entanto, esse programa não possui nenhuma implementação de segurança em sua arquitetura. Assim, decidiu-se estudar o protocolo QUIC e tentar integrá-lo ao programa *chownat* sem alterar seu comportamento original.

Neste capítulo são explicadas as etapas de desenvolvimento da aplicação. A linguagem escolhida para o desenvolvimento foi **C**, tendo como principais motivos seu alto desempenho e sua compatibilidade com os dispositivos (neste contexto, roteadores) utilizados no projeto.

A explicação inicia pelas bibliotecas/*APIs* utilizadas na implementação e os motivos de sua escolha, seguida pela estrutura do programa e, por fim, pelo resultado esperado: com foco no que o programa deve entregar durante sua execução.

4.1 Biblioteca *MsQuic*

Para a implementação do protocolo QUIC, optou-se pela utilização de bibliotecas *open-source* que já oferecessem as funcionalidades do protocolo prontas para uso, uma vez que desenvolver todas as funções do protocolo desde o início aumentaria significativamente a complexidade do projeto, bem como o tempo necessário para a sua implementação.

Por esse motivo, foi escolhida a biblioteca *MsQuic* [https://github.com/microsoft/msquic], cuja criação e desenvolvimento foram realizados pela *Microsoft*. A biblioteca foi escrita em *C*, com foco em desempenho e compatibilidade com múltiplas plataformas, como *Windows* e *Linux*.

A *MsQuic* possui como diferencial o suporte ao *TLS 1.3*, permitindo que as conexões estabelecidas sejam seguras sem a necessidade de bibliotecas externas de criptografia. Outros recursos, como controle de congestionamento, retransmissão de pacotes e *keep-alive*, são gerenciados internamente, simplificando o desenvolvimento de aplicações de rede.

4.2 Estrutura do Código

O código foi estruturado seguindo o modelo *cliente-servidor*, em que um programa denominado ***quic_server.c*** atua como servidor e o programa denominado ***quic_client.c*** atua como cliente.

4.2.1 Programa: *quic_server.c*

O código ***quic_server.c*** inicia com a inclusão da biblioteca *MsQuic* e de seus respectivos *headers*, permitindo o uso das diversas funções disponibilizadas para comunicação via protocolo QUIC. Após os *includes*, são definidas algumas configurações básicas, como a definição da porta para a conexão QUIC (50072), além dos caminhos para os arquivos de certificado e chave privada que serão utilizados para a autenticação *TLS*.

Além disso, são inicializados os principais objetos da biblioteca, como: ***Registration***, ***Configuration***, ***Listener*** e ***QuicStream***, descritos brevemente a seguir:

- ***Registration***: É o primeiro objeto a ser criado e agrupa todas as conexões da aplicação, permitindo que a biblioteca funcione em um "contexto global da aplicação"dentro do *MsQuic* — ou seja, ela atua no agrupamento e no gerenciamento de todos os recursos compartilhados por conexões, *listeners* e *streams*.
- ***Configuration***: Define os parâmetros de configuração de uma conexão QUIC, como certificados *TLS*, controle de congestionamento e políticas de segurança. É derivado de um *Registration* e é utilizado tanto pelo cliente quanto pelo servidor para iniciar as conexões seguras.
- ***Listener***: Utilizado apenas pelo servidor, o *Listener* escuta novas conexões QUIC em uma porta definida. Ao receber uma nova conexão, dispara um evento que pode ser tratado para aceitar e configurar a conexão com base na *Configuration* associada.
- ***QuicStream***: Representa um fluxo de dados dentro de uma conexão QUIC. O QUIC suporta múltiplos fluxos independentes por conexão. Cada fluxo pode ser usado para envio e recebimento bidirecional de dados e possui seus próprios *callbacks* para tratar eventos como recebimento de dados, abertura e encerramento.

O servidor também implementa *callbacks* importantes, como: a exibição dos dados recebidos do cliente no terminal; o *callback* de conexão, que lida com o início e encerramento de conexões; e o *callback* de *listener*, que configura novas conexões recebidas. Durante a inicialização do programa, o servidor carrega seus certificados e configura os parâmetros de segurança necessários para aceitar conexões. Um componente essencial do funcionamento do servidor é o uso de *sockets TCP* locais. Ao receber dados de uma *stream QUIC*, o servidor aceita conexões *TCP* locais e encaminha o conteúdo recebido

via QUIC para esse *socket TCP*, estabelecendo uma ponte entre o transporte QUIC e uma aplicação local baseada em *TCP*. Esse mecanismo permite que o servidor atue como um túnel *TCP over QUIC*, recebendo dados de clientes remotos via QUIC e direcionando-os para serviços locais via *TCP*.

4.2.2 Programa: *quic_client.c*

Por fim, temos o código ***quic_client.c***, onde é implementado o lado cliente da comunicação. Sua estrutura é semelhante à do servidor, pois utiliza as mesmas variáveis globais para o controle dos objetos da biblioteca *MsQuic*, bem como suas funções de inicialização, configuração e encerramento da conexão.

A principal diferença presente nesse código está na definição das configurações de portas e do endereço remoto, que são utilizadas para estabelecer a conexão com a parte servidora. O cliente possui duas *callbacks*: uma para tratar eventos do *stream*, imprimindo na tela os dados recebidos do servidor; e outra para tratar a conexão, iniciando automaticamente um *stream* assim que a conexão é estabelecida com sucesso.

Além disso, o cliente, assim como o servidor, implementa um *socket TCP* local na porta 44444, aceitando conexões de aplicações locais. Os dados recebidos dessas conexões *TCP* são imediatamente retransmitidos ao servidor remoto através do canal QUIC, enquanto dados provenientes do servidor, recebidos via QUIC, são encaminhados de volta à respectiva aplicação *TCP* local conectada. Esse mecanismo faz com que cliente e servidor atuem em conjunto como um túnel *TCP over QUIC*, permitindo a integração transparente entre aplicações *TCP* locais e serviços remotos acessíveis via QUIC. O código também gerencia o ciclo de vida dessas conexões, tratando desconexões de clientes *TCP*, possíveis erros de transmissão e realizando a limpeza apropriada dos recursos alocados.

Ambos os códigos estão disponíveis em https://github.com/ipaladinjr/MsQuic_TCC.

4.3 Comportamento Esperado

Os programas, ao serem compilados e executados, devem, por padrão, funcionar da seguinte forma:

- Após ser inicializado, o servidor carrega seu certificado e chave *TLS*, configura o *MsQuic* e passa a escutar por conexões QUIC na porta 50072 e conexões *TCP* na porta 8080.
- Durante a inicialização, o servidor exibe mensagens como:
  - `[INIT] Starting QUIC relay server...`

  - `[QUIC] msquic API and TLS configuration loaded successfully.`
  - `[MAIN] Ready: Accepting TCP on 127.0.0.1:8080, QUIC on port 50072`
- Em seguida, inicializa-se o cliente, que por padrão conecta-se a `127.0.0.1:50072` e fica ouvindo por conexões *TCP* locais na porta 44444.
- O cliente também exibe mensagens de inicialização, como:
  - `[INIT] Starting QUIC relay client...`
  - `[QUIC] msquic API and credentials loaded successfully.`
  - `[MAIN] Ready: Accepting TCP on 127.0.0.1:44444, QUIC to 127.0.0.1:50072`
- Se ambos os programas forem executados nessa ordem, ocorrerá a conexão entre o servidor e o cliente e a abertura dos canais de *stream* para a troca de dados. Isso será informado em ambos os programas por meio de mensagens de conexão. Por exemplo, no cliente pode-se observar:
  - `[QUIC] Connected to server! Connection is stable and ready.`
  - `[QUIC] Creating new stream...`
  - `[QUIC] New stream created and started successfully.`
- O mesmo ocorre no servidor, que ao receber a tentativa de conexão do cliente, exibe o recebimento de uma nova conexão QUIC e dá reconhecimento de uma nova *stream* de dados junto a essa conexão:
  - `[QUIC] New QUIC connection received. Setting configuration and callback handler.`
  - `[QUIC] Connection established (client handshake complete).`
  - `[QUIC] *** PEER_STREAM_STARTED EVENT RECEIVED! ***`
  - `[QUIC] Stream callback handler set successfully.`
- Com isso, temos a conclusão do processo de estabelecimento da conexão entre cliente e servidor. Após essa etapa, é possível realizar testes de comunicação, enviando mensagens entre as aplicações com o auxílio de ferramentas como o *netcat*. Por exemplo, pode-se conectar à porta *TCP* local do cliente (`nc 127.0.0.1 44444`) e do servidor (`nc 127.0.0.1 8080`), e as mensagens digitadas serão encaminhadas automaticamente para o outro lado, através do túnel QUIC estabelecido entre ambos.

5. RESULTADOS OBTIDOS

Neste capítulo, é realizada a etapa de coleta de dados visando validar o funcionamento e o desempenho da aplicação desenvolvida. São apresentados cenários de teste voltados à verificação da troca de mensagens entre dois dispositivos em diferentes contextos. Durante a execução, foram monitorados aspectos como o estabelecimento da conexão, o envio e a recepção de dados, bem como o comportamento das funções de *callback* definidas para os eventos da conexão e dos *streams*. Também foram utilizados *softwares* de captura de pacotes para validar se as mensagens trocadas entre o cliente e o servidor estão sendo corretamente encaminhadas por meio do *relay* QUIC e se seguem as definições estruturais do protocolo.

5.1 Teste utilizando *localhost* (127.0.0.1)

O primeiro teste realizado foi com ambos os programas em sua forma *default*, em uma única máquina, em terminais distintos. O objetivo desse teste foi validar as funcionalidades dos programas, verificando se a implementação estava correta quanto à criação dos *sockets TCP*, à configuração da biblioteca *MsQuic* e à comunicação utilizando as *streams* do protocolo QUIC.

Com a execução dos códigos, obteve-se, conforme apresentado na Figura Figura 5.1, a confirmação das diversas operações realizadas, tanto no terminal no qual o servidor estava em execução quanto naquele que executava o cliente. Observa-se que ambos os programas inicializaram corretamente todos os componentes necessários, incluindo o carregamento dos certificados, a configuração dos *listeners* QUIC e dos *sockets TCP* locais, além da criação dos *handlers* de conexão e de *stream*. Como alternativa para comprovar que o programa realmente conectou e que o protocolo QUIC foi utilizado na comunicação, foi realizada uma captura do tráfego na interface de *loopback*, apresentada na Figura Figura 5.2.

A análise da captura revela, inicialmente, os pacotes referentes ao processo de *handshake* do QUIC, essenciais para o estabelecimento da comunicação segura. Na sequência, observa-se uma série de pacotes rotulados como *“Protected Payload”*, indicando que a conexão foi estabelecida com sucesso e que o canal criptografado está ativo. Destaca-se que esses pacotes aparecem mesmo antes do envio de qualquer dado de aplicação pelo *socket TCP*, evidenciando que o próprio protocolo QUIC realiza trocas de mensagens de controle e manutenção do canal. Dessa forma, os testes comprovam tanto o correto funcionamento da lógica de inicialização dos programas quanto a efetiva utilização do protocolo QUIC para a troca segura de dados.

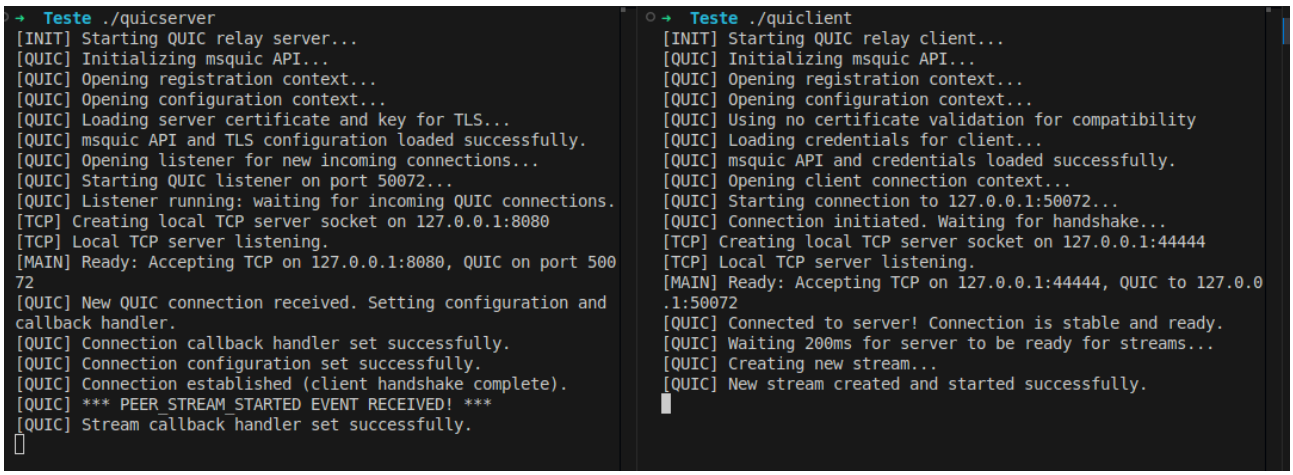

Figura 5.1 – Capturas de telas de execução de ambos os programas - Teste *localhost*.

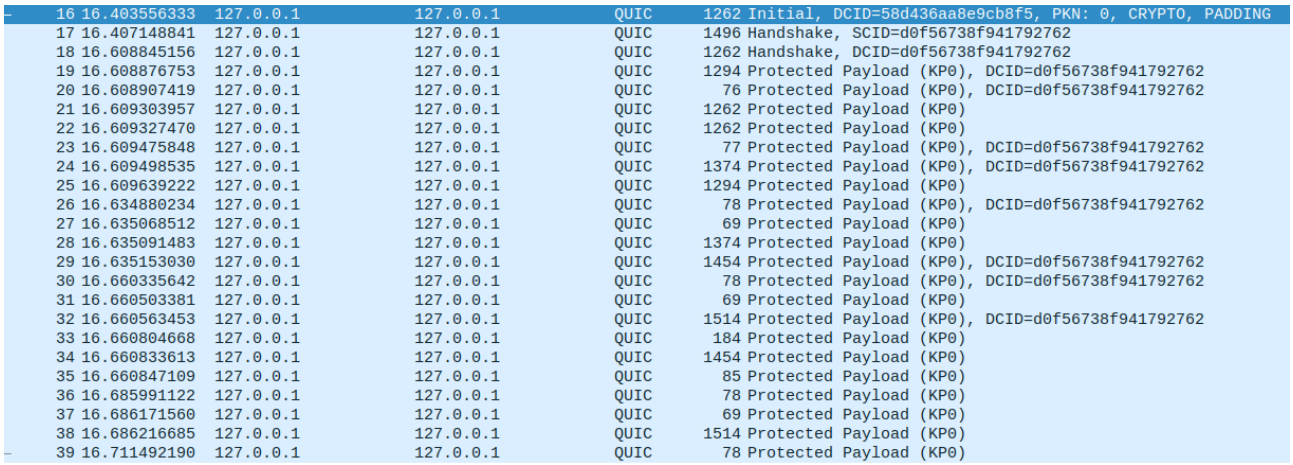

Figura 5.2 – Captura do *Wireshark* - Teste *localhost*.

5.2 Teste utilizando 2 máquinas de IPs diferentes com troca de mensagens

Nas Figuras 5.3 e 5.4, mais especificamente nas regiões destacadas pelo retângulo de cor verde, ilustram o estabelecimento bem-sucedido da conexão. Nota-se, pelos endereços *IP* apresentados e pelas mensagens de *log*, que se trata de dois dispositivos distintos, configurados para operar na mesma rede local, porém com endereços diferentes.

Nas saídas dos terminais do cliente (Figura 5.3) e do servidor (Figura 5.4), observa-se que ambos os programas inicializaram corretamente, configurando os contextos *MsQuic*, os *sockets TCP* locais e os respectivos manipuladores de eventos. Os *logs* detalham a aceitação de conexões *TCP* locais, a abertura de *streams* QUIC e o encaminhamento de dados entre os dispositivos. Podemos ver essa troca de informação entre os dispositivos, nas regiões destacadas pelo retângulo de cor vermelho, onde se verifica que dados trafegaram com sucesso do cliente para o servidor e vice-versa, com mensagens como *“Read 11 bytes*

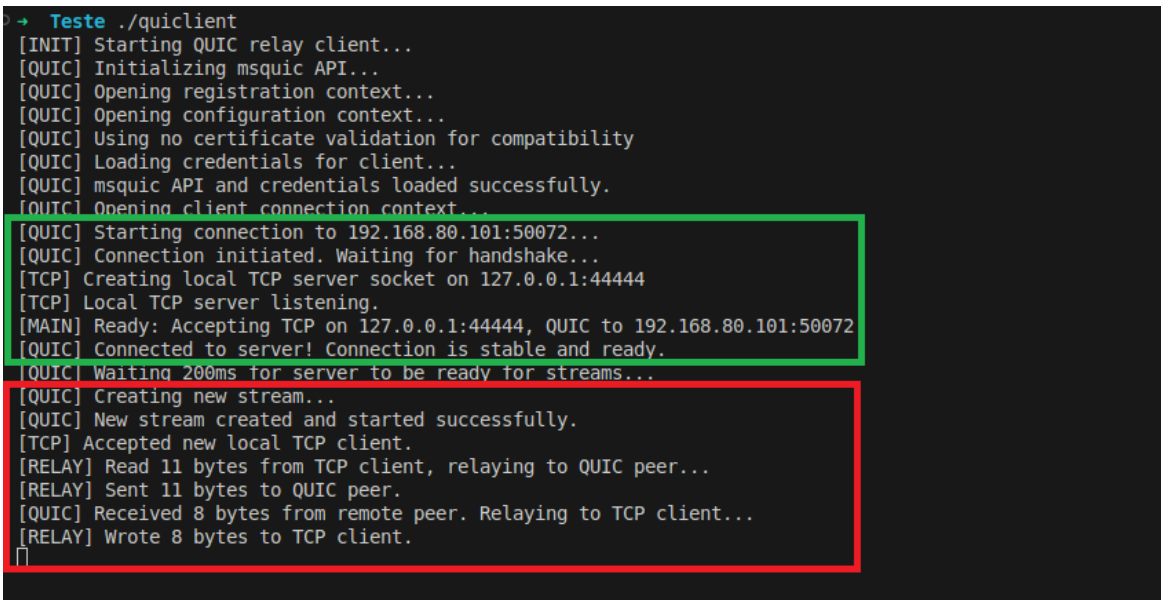

Figura 5.3 – Terminal *Client* - 192.168.80.129.

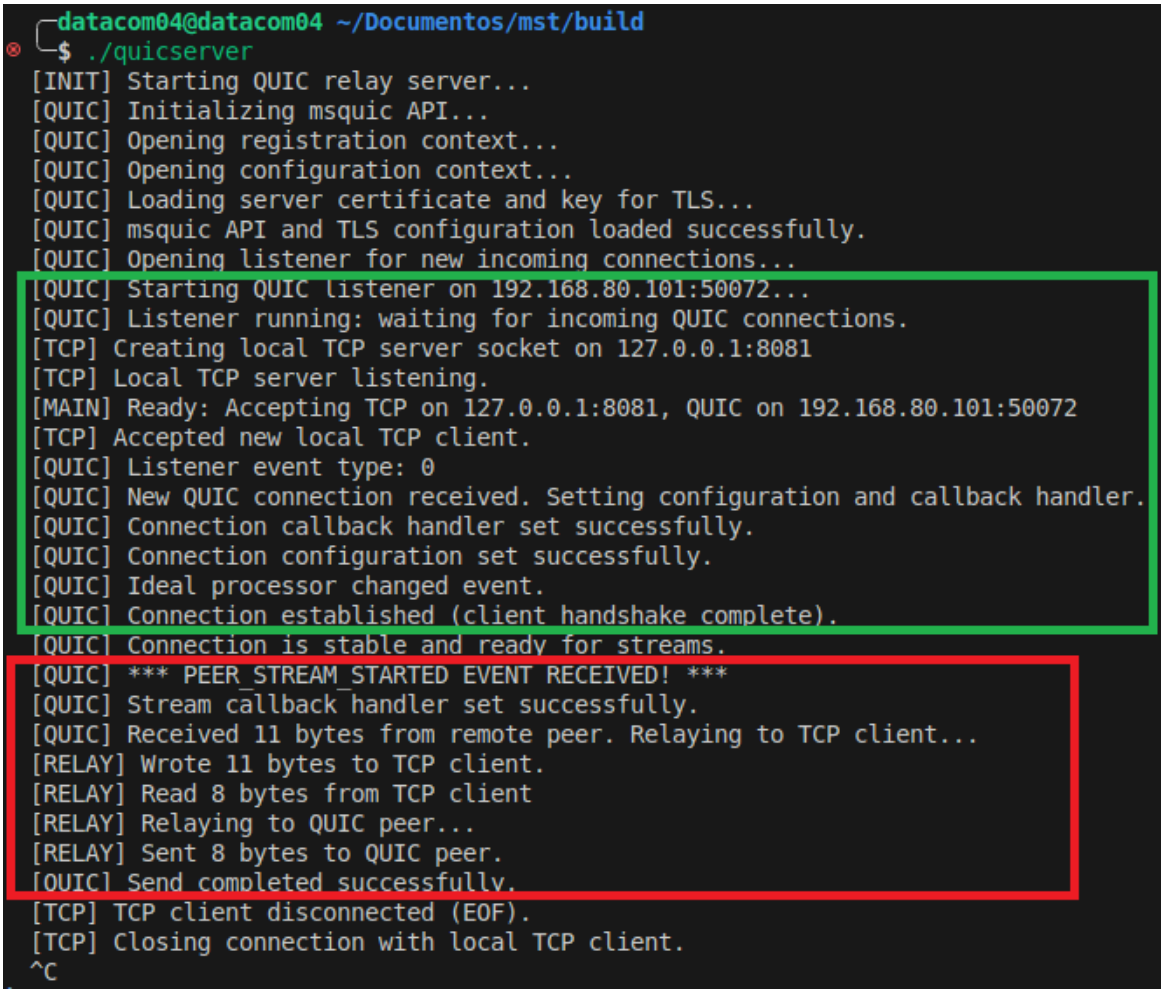

Figura 5.4 – Terminal *Server* - 192.168.80.101.

from TCP client, relaying to QUIC peer...” e *“Received 8 bytes from remote peer. Relaying to TCP client...”*, comprovando o funcionamento bidirecional do túnel *TCP over QUIC*.

A captura de pacotes exibida na Figura 5.5 demonstra o processo de estabelecimento da conexão QUIC entre os dois dispositivos. Os primeiros pacotes correspondem ao *handshake* do protocolo QUIC, seguidos por uma sequência de pacotes do tipo *"Protected Payload"*. Isso confirma que a comunicação segura foi estabelecida e que o canal criptografado está ativo entre os dispositivos.

No.	Time	Source	Destination	Protocol	Length	Info
283	17.921374244	192.168.80.129	192.168.80.101	QUIC	1262	Initial, DCID=5274bb9d71b0e79c, PKN: 0, CRYPTO, PADDING
284	17.925247201	192.168.80.101	192.168.80.129	QUIC	1262	Handshake, SCID=637b2b2f93e81839ee
285	17.925276258	192.168.80.101	192.168.80.129	QUIC	392	Handshake, SCID=637b2b2f93e81839ee
292	18.925207085	192.168.80.101	192.168.80.129	QUIC	1262	Handshake, SCID=637b2b2f93e81839ee
293	18.927591886	192.168.80.129	192.168.80.101	QUIC	1262	Initial, DCID=5274bb9d71b0e79c, PKN: 1, CRYPTO, PING, PADDING
294	18.952820206	192.168.80.101	192.168.80.129	QUIC	1262	Initial, SCID=637b2b2f93e81839ee, PKN: 5, ACK, PADDING
306	19.149443985	192.168.80.129	192.168.80.101	QUIC	2516	Handshake, DCID=637b2b2f93e81839ee
307	19.149800853	192.168.80.101	192.168.80.129	QUIC	1262	Protected Payload (KP0)
308	19.149841664	192.168.80.101	192.168.80.129	QUIC	1294	Protected Payload (KP0)
309	19.152662561	192.168.80.129	192.168.80.101	QUIC	77	Protected Payload (KP0), DCID=637b2b2f93e81839ee
310	19.152662988	192.168.80.129	192.168.80.101	QUIC	1294	Protected Payload (KP0), DCID=637b2b2f93e81839ee
311	19.152787723	192.168.80.101	192.168.80.129	QUIC	1374	Protected Payload (KP0)
312	19.177974830	192.168.80.101	192.168.80.129	QUIC	69	Protected Payload (KP0)
313	19.179837251	192.168.80.129	192.168.80.101	QUIC	78	Protected Payload (KP0), DCID=637b2b2f93e81839ee
314	19.179837636	192.168.80.129	192.168.80.101	QUIC	1374	Protected Payload (KP0), DCID=637b2b2f93e81839ee
315	19.179927455	192.168.80.101	192.168.80.129	QUIC	1454	Protected Payload (KP0)
316	19.205115778	192.168.80.101	192.168.80.129	QUIC	69	Protected Payload (KP0)
317	19.208312570	192.168.80.129	192.168.80.101	QUIC	1454	Protected Payload (KP0), DCID=637b2b2f93e81839ee
318	19.209748890	192.168.80.129	192.168.80.101	QUIC	77	Protected Payload (KP0), DCID=637b2b2f93e81839ee
319	19.209792422	192.168.80.101	192.168.80.129	QUIC	1514	Protected Payload (KP0)
320	19.233969287	192.168.80.101	192.168.80.129	QUIC	69	Protected Payload (KP0)
321	19.236126103	192.168.80.129	192.168.80.101	QUIC	1514	Protected Payload (KP0), DCID=637b2b2f93e81839ee
322	19.239515657	192.168.80.129	192.168.80.101	QUIC	77	Protected Payload (KP0), DCID=637b2b2f93e81839ee
323	19.261695003	192.168.80.101	192.168.80.129	QUIC	69	Protected Payload (KP0)

Figura 5.5 – Captura *Wireshark* - Teste com 2 dispositivos.

Essas mensagens, que indicam o envio e o recebimento de *bytes*, referem-se à conexão local estabelecida por meio do comando `nc` (*NetCat*), utilizando como parâmetro, em ambos os dispositivos, o endereço do *socket TCP* local correspondente. Com a conexão estabelecida dessa forma, foi possível enviar mensagens entre os dispositivos, que, ao serem transmitidas para os respectivos *sockets*, foram automaticamente encaminhadas pelo túnel QUIC e recebidas no outro extremo.

A Figura Figura 5.6 ilustra esse procedimento: nela, observa-se a execução do comando `nc 127.0.0.1 44444`, que conecta a aplicação local ao *socket TCP* do cliente. Em seguida, a mensagem `"Oi William"` é digitada e enviada, sendo recebida e respondida com `"Oi Ivan"`, demonstrando que a comunicação bidirecional foi realizada com sucesso por meio do túnel *TCP over QUIC* implementado. Esse teste comprova que os dados enviados pelo *socket TCP* local de um dispositivo são corretamente transmitidos via QUIC ao dispositivo remoto e vice-versa, validando a funcionalidade de encaminhamento de dados entre as aplicações.

```
→  ~ nc 127.0.0.1 44444
Oi William
Oi Ivan
```

Figura 5.6 – *netcat* - cliente.

Esses resultados evidenciam que a solução proposta não só realiza corretamente o estabelecimento de conexões em ambiente real, como também garante a transmissão de dados entre equipamentos distintos, validando o funcionamento da ponte *TCP over QUIC* implementada.

6. CONCLUSÃO E TRABALHOS FUTUROS

Este trabalho atendeu ao que foi proposto no título: “Protocolo QUIC para securizar a conexão de Roteadores”. A pesquisa iniciou-se visando entender e verificar a viabilidade do protocolo, para uma possível inclusão deste na aplicação desenvolvida pelo projeto. Os resultados obtidos ao longo da pesquisa permitiram uma compreensão mais aprofundada do protocolo, possibilitando o desenvolvimento de um programa capaz de demonstrar seu funcionamento. Além disso, o código em C desenvolvido foi parcialmente integrado a outro código considerado essencial para a aplicação, aprimorando um ponto relevante inicialmente não considerado: a segurança dos dados. Assim, este trabalho representa uma base sólida para futuras modificações que possam ser implementadas.

Em conclusão, este trabalho representa um avanço na área de redes, ao utilizar um protocolo relativamente novo de forma não usual, integrando-o a uma aplicação fora dos contextos em que geralmente é empregado, como na configuração e monitoramento de dispositivos de rede.

Este TCC proporcionou ao autor um aprofundamento em diversos conhecimentos relacionados à área de Engenharia de Computação. Durante o desenvolvimento do projeto, foram adquiridos conhecimentos sobre redes e suas diferentes áreas de atuação, bem como sobre os diversos contextos de aplicação de cada protocolo. Esses conhecimentos são relevantes para a formação do autor como Engenheiro de Computação, capacitando-o a enfrentar desafios e a se adaptar às diversas áreas abordadas pelo curso.

Trabalhos futuros incluem:

1. **Implementar** a lógica dos *sockets UDP*, que permitem a comunicação direta com os *sockets* TCP locais, com o objetivo de realizar o método de *hole-punch* através do *NAT*, viabilizando a comunicação remota entre dois dispositivos.
2. Após a implementação e a realização de testes que comprovem o funcionamento do método de *hole-punch* mencionado no item anterior, realizar a tradução dos códigos desenvolvidos em **C** para **Dart**, com o objetivo de integrá-los à aplicação desenvolvida e aos dispositivos utilizados no projeto.
3. Realizar a integração dos códigos em seus respectivos contextos: o programa denominado **cliente** é integrado à aplicação desenvolvida, enquanto sua contrapartida, o **servidor**, é portada diretamente para os roteadores, por meio de compiladores cruzados.
4. Após a tradução dos códigos para **Dart** e a integração destes na aplicação e nos dispositivos, realizar testes de produção, nos quais serão avaliadas a funcionalidade e a performance dos códigos, considerando se o código, após a tradução, continua executando sua funcionalidade principal e como se comporta no ambiente da aplicação, com foco em verificar se sua integração não acarretou falhas ou quebras de contexto em implementações anteriores.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Ahmed G. Wadday, Salim M. Wadi, H. J. M. A. A. A. “Study of WiMAX Based Communication Channel Effects on the Ciphered Image Using AES Algorithm”, *International Journal of Applied Engineering Research*, vol. 13–8, 2018, pp. 6009–6018.

[2] CloudflareTeam. “O que é UDP?” Capturado em: https://www.cloudflare.com/pt-br/learning/ddos/glossary/user-datagram-protocol-udp/, 2022.

[3] CloudflareTeam. “What is Transport Layer Security (TLS)?” Capturado em: hhttps://www.cloudflare.com/pt-br/learning/ssl/what-happens-in-a-tls-handshake/, 2022.

[4] DARPA INTERNET PROGRAM. “Transmission Control Protocol – RFC 793”. Capturado em: https://www.ietf.org/rfc/rfc793.txt, 1981.

[5] DARPA INTERNET PROGRAM. “Asymmetric Cryptography – Introduction”. Capturado em: https://www.noser.com/techblog/asymmetric-cryptography-introduction/, 2016.

[6] DIO TEAM. “Você conhece o protocolo HTTP e suas versões?” Capturado em: https://www.dio.me/articles/voce-conhece-o-protocolo-http-e-suas-versoes/, 2023.

[7] Geeks for Geeks. “Advanced Encryption Standard (AES)”. Capturado em: https://www.geeksforgeeks.org/advanced-encryption-standard-aes/, 2024.

[8] Geeks for Geeks. “Transport Layer Security (TLS)”. Capturado em: https://www.geeksforgeeks.org/transport-layer-security-tls/, 2024.

[9] Geeks for Geeks. “What is HTTP ?” Capturado em: https://www.geeksforgeeks.org/what-is-http//, 2024.

[10] Geoff Huston. “tcp+tlsxquic”. Capturado em: https://blog.apnic.net/2022/11/03/comparing-tcp-and-quic//, 2022.

[11] Huston, G. “A quick look at QUIC”. Capturado em: https://labs.apnic.net/index.php/2019/03/04/a-quick-look-at-quic/, 2019.

[12] Internet Engineering Task Force (IETF). “RSA Cryptography Specifications Version 2.2”. Capturado em: https://datatracker.ietf.org/doc/html/rfc8017, 2016.

[13] Iyengar, J.; Thomson, M. “QUIC: A UDP-Based Multiplexed and Secure Transport – RFC 9000”. Capturado em: https://datatracker.ietf.org/doc/rfc9000/, 2019.

[14] Iyengar, J.; Thomson, M. “QUIC: A UDP-Based Multiplexed and Secure Transport”. Capturado em: https://datatracker.ietf.org/doc/html/rfc9000, 2021.

[15] Langley, A.; et al.. “The QUIC Transport Protocol: Design and Internet-Scale Deployment”. In: ACM Special Interest Group on Data Communication, SIGCOMM 2017, 2017, pp. 183–196, https://doi.org/10.1145/3098822.3098842.

[16] Robert Haynes. “Conheça a rede e criptografia QUIC no NGINX”. Capturado em: https://www.f5.com/pt_br/company/blog/nginx/primer-quic-networking-encryption-in-nginx/, 2023.

[17] Samy Kamkar. “chownat”. Capturado em: https://samy.pl/chownat/, 2004.

[18] The Wireshark team. “Wireshark”. Capturado em: https://www.wireshark.org/, 1998.

[19] W3Tech. “Usage statistics of QUIC for websites”. Capturado em: https://w3techs.com/technologies/details/ce-quic, 2025.