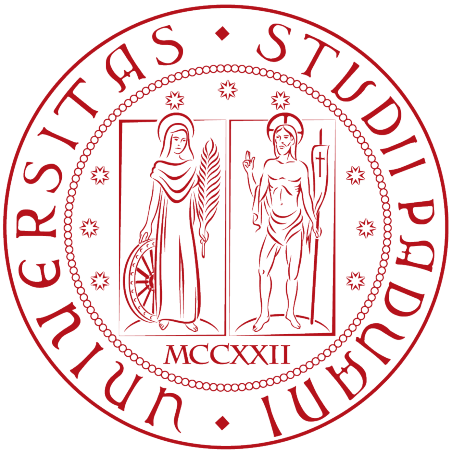
Università degli Studi di Padova

Dipartimento di Matematica "Tullio Levi-Civita"

Corso di Laurea in Informatica



Valutazione della Resilienza di QUIC: Manipolazione Selettiva del Traffico

Tesi di laurea

Relatore
Prof. Alessandro Galeazzi

Correlatore
Dott. Enrico Bassetti

Laureando
Giovanni Menon

Matricola 2034301

Anno Accademico 2023-2024

"If something is important enough, you should try. Even if the probable outcome is failure."

— Elon Musk

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage, della durata complessiva di trecentoventi ore, dal laureando Giovanni Menon presso l'Università degli studi di Padova. Il tirocinio è stato condotto sotto la guida del Prof. Alessandro Galeazzi e con la collaborazione del Dott. Enrico Bassetti. Il Prof. Davide Bresolin ha ricoperto il ruolo di tutor accademico e referente interno al Consiglio del Corso di Studio.

Questa tesi riguarda l'analisi del protocollo *Quick UDP Internet Connections (QUIC)* nel contesto delle reti moderne, con particolare attenzione alle problematiche legate alla tariffazione del traffico dati. Lo studio ha esplorato alcune possibili strategie che potrebbero essere sfruttate da un malintenzionato per manipolare artificialmente il traffico mobile. I risultati ottenuti hanno evidenziato come queste strategie possano indurre un incremento del traffico dati per l'utente vittima, aumentando di conseguenza i suoi consumi dati e i suoi relativi costi.

Il tirocinio si è suddiviso in due parti. La prima dedicata ad un'analisi approfondita del protocollo *QUIC*, esaminando non solo lo stato dell'arte attuale e le tecnologie associate, ma anche la sua logica intrinseca e il suo funzionamento. La seconda parte, invece, si è concentrata sulla progettazione e realizzazione degli esperimenti volti a testare le strategie identificate, terminando nella raccolta e nell'analisi dei risultati ottenuti.

Ringraziamenti

Giunto alla fine di questo percorso accademico, desidero esprimere la mia profonda riconoscenza verso tutti coloro che mi hanno sostenuto e che ho avuto la fortuna di incontrare.

In primis, desidero esprimere la mia più profonda gratitudine al Prof. Alessandro Galeazzi, relatore di questa tesi, per il suo prezioso supporto durante l'intero processo di stesura. Un sentito ringraziamento va anche al Dott. Enrico Bassetti, la cui supervisione durante il tirocinio e i cui consigli si sono rivelati una guida fondamentale per la realizzazione di questo lavoro.

Un ringraziamento speciale va ai miei genitori, Maria Teresa e Doriano, e a mio fratello Francesco, per avermi sostenuto, incoraggiato e sopportato durante questo percorso. Un grazie particolare ai miei genitori, i cui sacrifici mi hanno dato l'opportunità di intraprendere e completare questo percorso.

Un caloroso ringraziamento va alle mie nonne, Angela e Luciana, che con il loro affetto mi sono state vicine durante questi tre anni di studi. Un sincero ringraziamento anche ai miei zii e alle mie zie per il loro prezioso e costante sostegno.

Un grazie speciale va ai miei più cari amici Jacopo, Davide e Marco, presenze costanti in ogni momento di questo percorso. La vostra vicinanza, il vostro sostegno e la vostra capacità di spronarmi a dare il meglio di me sono stati fondamentali.

Un grazie a tutti coloro che ho conosciuto lungo questo percorso. In particolare ringrazio i ragazzi di CodingCowboys e i miei amici del Concilio per essere stati una compagnia costante in questi anni di studio.

Padova, Settembre 2024 Giovanni Menon

Indice

Elenco delle figure

Elenco delle tabelle

Capitolo 1

Introduzione

1.1 Motivazione

L'avvento di Internet e la continua evoluzione delle tecnologie di comunicazione ha trasformato radicalmente il modo in cui le persone interagiscono, lavorano e accedono alle informazioni. Tuttavia, l'aumento esponenziale del traffico e la crescente complessità delle reti moderne comportano nuove sfide nella gestione e tariffazione del traffico. Attualmente, le infrastrutture si basano su piani tariffari che dipendono da soglie prestabilite o sul consumo effettivo di dati, il cui calcolo è affidato agli operatori di rete. Questo approccio, tuttavia, introduce una serie di nuove problematiche. Un esempio significativo è rappresentato dalla possibilità che un malintenzionato potrebbe sfruttare le caratteristiche intrinseche di un protocollo di rete o del sistema di comunicazione per manipolare artificialmente il consumo dati degli utenti senza che questi ne siano consapevoli. Tale manipolazione comporta un aumento ingiustificato dei costi o l'esaurimento prematuro delle risorse dati disponibili. In questo contesto, questo studio si propone di analizzare in dettaglio alcune delle possibili strategie che possono essere adottate dai malintenzionati, valutandone gli impatti potenziali e studiandone gli effetti sul consumo dati di una vittima. Questa analisi è importante per esplorare la portata delle minacce e i potenziali rischi connessi legati a questa tipologia di attacco.

1.2 Organizzazione del testo

Di seguito, viene presentata la struttura del documento :

Il secondo capitolo presenta quanto trovato di simile nella letteratura attuale;

Il terzo capitolo approfondisce il background e illustra l'idea del progetto;

Il quarto capitolo descrive dettagliatamente l'ambiente di sviluppo e presenta i singoli esperimenti condotti;

Il quinto capitolo riassume e analizza i risultati ottenuti dagli esperimenti;

Il sesto capitolo presenta le conclusioni del lavoro e propone possibili sviluppi futuri.

Capitolo 2

Related Works

Sia Langley et al. che Rüth et al. nei loro studi del 2017 e 2018 presentano una descrizione dettagliata del protocollo *QUIC* concentrandosi sulla sua scalabilità e sugli effetti che esso ha sulle reti moderne [1, 2]. I loro lavori affrontano in dettaglio le motivazioni principali che hanno portato allo sviluppo di *QUIC* e analizzano come il protocollo sia stato progettato per supportare servizi su scala globale, come *YouTube* e *Google*.
Lychev et al., nel loro studio, analizzano le performance di *QUIC* evidenziando un compromesso tra la riduzione della latenza e la garanzia di sicurezza. Gli autori, tramite un modello di sicurezza specifico, hanno dimostrato che il protocollo non garantisce la *forward secrecy* tradizionale offerta da *TLS*. Lo studio mostra inoltre come una serie di attacchi semplici possano compromettere i vantaggi di *QUIC*, sottolineando come i meccanismi utilizzati per aumentare la velocità del protocollo siano anche la causa di diverse debolezze.
Gli studi, condotti nel 2022 e 2023, di Chatzoglou et al. e Gbur e Tschorsch offrono un'analisi più dettagliata e approfondita della sicurezza del protocollo *QUIC*, mettendo in luce come esso sia vulnerabile a numerosi attacchi e presenti diverse criticità in termini di sicurezza. Nello studio di Gbur e Tschorsch vengono analizzate le vulnerabilità di *QUIC* sugli attacchi di *client-side request forgery*. Gli autori dimostrano, attraverso diverse modalità di *request forgery*, che il protocollo è vulnerabile a fenomeni di amplificazione del traffico, nonostante i limiti anti-amplificazione previsti nella specifica *RFC* del protocollo. I risultati del lavoro di Chatzoglou et al. sono particolarmente rilevanti nella letteratura, in quanto hanno portato all'identificazione di diverse vulnerabilità *zero-day* efficaci e pratiche. Gli autori hanno esaminato sei tra i più popolari *server* compatibili con *QUIC* utilizzando tecniche di *fuzz testing* per l'identificazione delle vulnerabilità. Questi test hanno evidenziato come le implementazioni a livello di produzione di *QUIC* non siano ancora sufficientemente pronte per le reti moderne [4, 5].
Nel 2014, Go et al. hanno pubblicato uno studio in cui vengono discusse alcune vulnerabilità delle reti mobili e della tariffazione del traffico dati per il protocollo *TCP* [6]. Questo lavoro ha sottolineato un importante problema di politiche relativo alla contabilizzazione dei dati da parte degli operatori. Lo studio rivela che le ritrasmissioni *TCP* possono essere facilmente sfruttate per manipolare la contabilizzazione del traffico cellulare. Questa scoperta non solo identifica una potenziale vulnerabilità nei sistemi di contabilizzazione del traffico dati, ma dimostra anche come i diversi operatori adottino politiche differenti per il conteggio del traffico dati.

A differenza degli studi citati, il nostro si concentra specificamente sul protocollo *QUIC* nel contesto delle reti mobili e sulla contabilizzazione del traffico dati. Il nostro studio trae ispirazione dal lavoro svolto da Go et al., che ha analizzato come alcune vulnerabilità del protocollo *TCP* possano essere sfruttate per manipolare la contabilizzazione del traffico dati nelle reti mobili [6]. Applicando un approccio simile, si esaminano alcune potenziali strategie che sfruttano *QUIC* per manipolare artificialmente il traffico dati. Ci si focalizza sulla possibilità che *QUIC*, come *TCP*, possa essere soggetto a problematiche di contabilizzazione.

Capitolo 3

Descrizione del Progetto

Questo capitolo introduce i preliminari tecnici e illustra il problema affrontato. La sezione 3.1 è dedicata alla descrizione del contesto tecnologico mentre la sezione 3.2 analizza le problematiche affrontate nello studio.

3.1 Scenario Tecnico

3.1.1 Modello ISO/OSI e Protocolli di Rete

I protocolli di rete sono una componente essenziale per la comunicazione tra dispositivi su una rete. Definiscono le regole e gli standard per il trasferimento dei dati, assicurando che le informazioni vengano trasmesse e comprese correttamente. Per facilitare la comprensione e l'organizzazione di questi protocolli, è utile fare riferimento al modello *ISO/OSI (Open Systems Interconnection)*. Questo standard architetturale suddivide i protocolli in sette livelli distinti elencati in Figura 3.1, ognuno con un compito specifico. Di seguito, viene riportata una descrizione del ruolo di ogni livello:

1. **Livello Fisico**: Il livello fisico si occupa della trasmissione effettiva dei bit attraverso il mezzo fisico della rete, in questo livello lavorano i *modem* e gli *hub*.

2. **Livello Data Link**: Il livello *data link* gestisce la comunicazione tra dispositivi sulla stessa rete fisica. Si occupa di garantire che il trasferimento mediante il livello fisico sia affidabile effettuando un controllo degli errori. A questo livello operano *switch* e i *bridge*.

3. **Livello Rete**: Il livello di rete gestisce l'instradamento ottimale dei pacchetti di dati tra reti diverse. Il protocollo principale che opera a questo livello è l'*Internet Protocol (IP)*.

4. **Livello Trasporto**: Il livello di trasporto controlla il flusso di dati e gestisce la segmentazione dei pacchetti. Garantisce la corretta trasmissione delle informazioni. I protocolli principali includono il *Transmission Control Protocol (TCP)* e il *User Datagram Protocol (UDP)*.

5. **Livello Sessione**: Il livello di sessione gestisce l'apertura, il controllo e la chiusura delle sessioni di comunicazione.

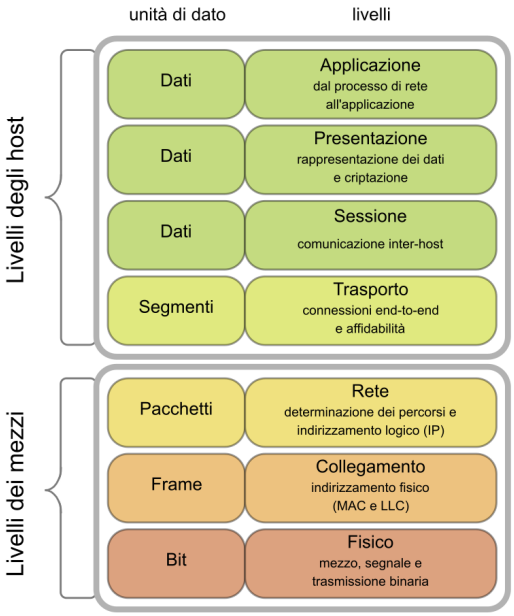


Figura 3.1: *Raffigurazione del modello ISO/OSI*

6. **Livello Presentazione**: Il livello di presentazione è responsabile della traduzione e della formattazione dei dati tra il formato utilizzato dalle applicazioni a quello della rete. Protocolli come *Secure Sockets Layer (SSL)* e *Transport Layer Security (TLS)* ne fanno parte.

7. **Livello Applicazione**: Il livello di applicazione fornisce un insieme di protocolli che operano a stretto contatto con le applicazioni. Protocolli come *HTTP*,*FTP* e *SMTP* ne fanno parte.

3.1.2 Protocolli di Trasporto Tradizionali

Nel panorama dei *protocolli di rete*, i *protocolli di trasporto TCP (Transmission Control Protocol)* e *UDP (User Datagram Protocol)* hanno svolto e svolgono tutt'ora un ruolo fondamentale sin dalla nascita di Internet. Questi protocolli sono stati la spina dorsale delle comunicazioni per decenni, supportando una vasta gamma di servizi e applicazioni.
Pur appartenendo alla stessa famiglia di protocolli, *TCP* e *UDP* sono stati concepiti con caratteristiche e obbiettivi distinti e vengono impiegati in base alle specifiche esigenze applicative. In particolare *TCP*, con la sua affidabilità e il suo controllo di flusso, ha svolto un ruolo fondamentale nelle comunicazioni che richiedevano l'integrità dei dati, mentre *UDP* ha trovato il suo spazio nei servizi che privilegiavano la velocità rispetto all'affidabilità. Tuttavia, con l'evoluzione delle nuove tecnologie, le limitazioni di questi protocolli sono diventate sempre più evidenti. Nella concezione di base di *TCP* e *UDP*, ideata agli inizi del 1970, non erano state previste le sfide delle reti moderne, caratterizzate da:

- Connessioni mobili e variabili;

- Necessità di ridurre la latenza;
- Proliferazione di dispositivi *IoT*[1];
- Requisiti di sicurezza sempre più vincolanti.

Queste nuove sfide hanno evidenziato una serie di problematiche nei protocolli. La consapevolezza di questi limiti ha portato alla ricerca di nuove soluzioni, cercando di superare le inefficienze pur mantenendo i punti di forza dei protocolli tradizionali. Questi studi hanno portato alla creazione di nuovi protocolli come *QUIC* ed a estensioni come *MPTCP*, che cercano di superare le limitazioni di *TCP* e *UDP* per far fronte alle sfide del mondo moderno, offrendo prestazioni migliori, maggiore sicurezza e flessibilità.

TCP (Transmission Control Protocol)

Il *Transmission Control Protocol (TCP)* è uno dei protocolli cardine su cui si basa la comunicazione su Internet. Dato il suo ruolo e la vasta gamma di funzioni che offre, un'analisi completa del suo funzionamento e della sua costituzione richiederebbe un'analisi approfondità che va oltre lo scopo di questa tesi. Pertanto, in questa sezione, ci si concentrerà solo su alcuni aspetti specifici del *TCP* che sono fondamentali per la comprensione del problema affrontato in questa tesi. In particolare, verranno esaminate nel dettaglio :

- Caratteristiche principali di una connessione *TCP*;
- Il processo di *Handshake*[2];
- Il meccanismo di Ritrasmissione;
- I metodi utilizzati per garantire la sicurezza dei dati.

Caratteristiche principali

Il *Transmission Control Protocol* si distingue come un protocollo orientato alla connessione. Ciò significa che, prima di qualsiasi scambio di dati, deve essere stabilita una connessione dedicata tra il *client*[3] e il *server*[4]. Tale peculiarità è alla base di molte delle sue funzionalità avanzate, tra cui:

Affidabilità Garantisce che i dati trasmessi vengano ricevuti correttamente.

Controllo di Errore Implementa un sistema di verifica della integrità dei dati tramite il meccanismo di *checksum*[5].

Controllo di Flusso e Congestione Utilizza il sistema delle *sliding window*[6] per ottimizzare il flusso di dati e diminuire il numero di segmenti ritrasmessi in caso di situazione di congestione.

[1] Internet of Things (IoT)
[2] *Handshake*
[3] *Client*
[4] *Server*
[5] *Checksum*
[6] *Sliding window*

Queste funzionalità si riflettono sulla struttura stessa di un segmento *TCP*, come si può vedere nella Figura 3.2. Di seguito, è riportata la descrizione di alcune sezioni di interesse per il problema analizzato:

- ***Source Port - Destination Port***: Identificano rispettivamente il numero della porta di origine e destinazione.
- ***Sequence Number***: Indica la posizione del primo segmento *TCP* all'interno del flusso a partire dall'*Initial Sequence Number (ISN)*, deciso durante la inizializzazione della connessione.
- ***Acknowledgement Number***: Se il *control bits ACK* è presente questo campo contiene il valore del prossimo *sequence number* che il ricevente del segmento si aspetta di ricevere.
- ***Control Bits***: Sono dei bit utilizzati per il controllo del protocollo.
  - ***URG***: Se impostato a 1 indica la presenza di dati urgenti.
  - ***ACK***: Se impostato a 1 indica che l'*acknowledgement number*[7] è valido.
  - ***PSH***: Se impostato a 1 indica che i dati devono essere elaborati dai livelli superiori.
  - ***RST***: Se impostato a 1 indica che la connessione non è più valida.
  - ***SYN***: Se impostato a 1 indica che il mittente vuole stabilire una connessione *TCP*.
  - ***FIN***: Se impostato a 1 indica che il mittente vuole terminare la connessione *TCP*.
- ***Checksum***: Valore utilizzato per verificare la validità del segmento. Si ottiene facendo il complemento a uno della somma complementare a uno a 16 bit dell'*header* e del *payload*.
- ***Data***: Contiene l'informazione da trasmettere.

Una connessione *TCP* è dunque identificata univocamente da quattro elementi:

$$[\text{Indirizzo IP sorgente}, \text{Porta sorgente}] \leftrightarrow [\text{Indirizzo IP destinazione}, \text{Porta destinazione}]$$

Questa struttura offre diversi vantaggi. Ad esempio, permette ad un singolo *server* di accettare più connessioni contemporaneamente da diversi *client*, e viceversa. Inoltre garantisce l'unicità di ogni connessione all'interno della rete.
Ciononostante, questa rigida definzione di una connessione presenta delle limitazioni significative, in quanto qualsiasi modifica a uno dei quattro elementi, come un cambio di porta o una migrazione dell'indirizzo IP, comporta la chiusura della connessione esistente. Questa limitazione diventa ancora più rilevante nel contesto delle comunicazioni moderne, dove situazioni di *multihoming*[8] e cambi frequenti di indirizzo IP sono sempre più comuni [7].

Questa problematica, come verrà sottolineato nelle prossime sezioni, è alla base di soluzioni come *MPTCP* e viene affrontata in modo indiretto anche in *QUIC*.

[7] *Acknowledgements*
[8] *Multihoming*

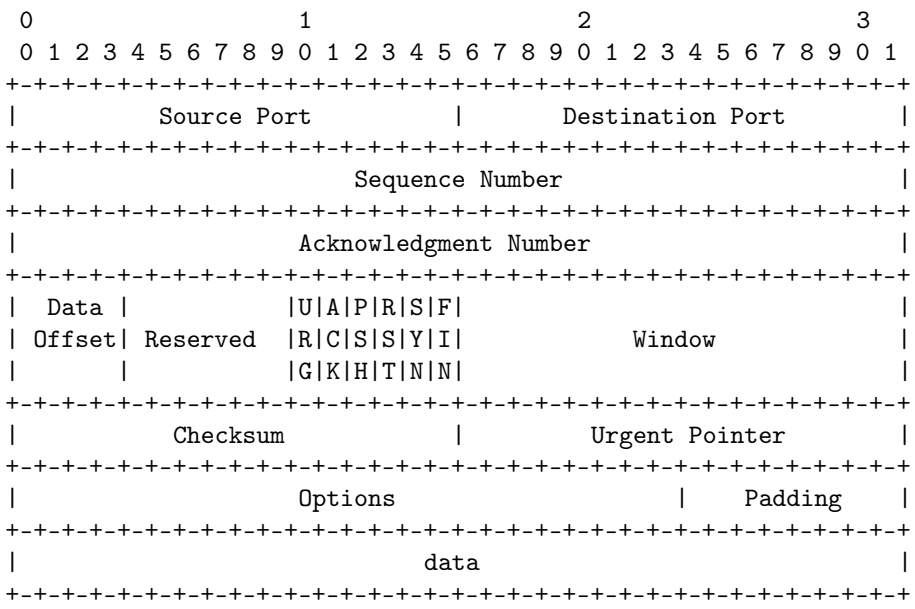


Figura 3.2: *Composizione di un segmento TCP*

Handshake

Uno dei punti cardine del *TCP* è la necessità di stabilire una connessione prima di qualsiasi scambio di dati. Questo processo, noto come *three-way handshake* ("stretta di mano in tre passaggi"), richiede lo scambio di tre messaggi tra *server* e *client*. La Figura 3.3 illustra questo procedimento.

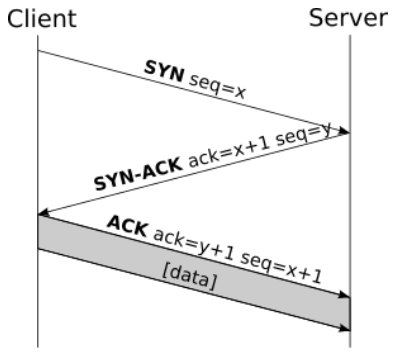


Figura 3.3: *Processo three-way handshake in TCP*

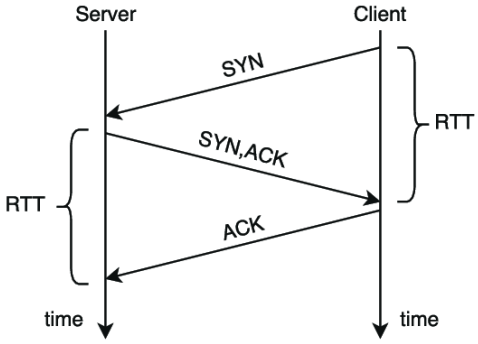


Figura 3.4: *Definizione di RTT (Round Trip Time)*

I passaggi coinvolti in questo processo sono i seguenti:

1. **Client invia un segmento SYN al Server**: Il segmento ha il campo *SYN* impostato a 1 e il suo *sequence number* contiene il valore x che rappresenta l'*ISN* del *Client*.

2. **Server invia un segmento SYN/ACK al Client**: Il *server* risponde con un segmento i cui campi *SYN* e *ACK* sono impostati a 1. Il suo *sequence*

number contiene un nuovo valore y che specifica l'*ISN* del *server* e il campo *Acknowledgement number* contiene il valore *x+1* del *client*.

3. **Client invia un segmento ACK al Server**: Il *client* risponde inviando un segmento il cui campo *ACK* è impostato a 1 e l'*acknowledgement number* è dato da *y+1*.

Oltre a stabilire la connessione, l'*handshake* riveste un ruolo fondamentale in vari aspetti della comunicazione *TCP*. Non solo permette di sincronizzare i *sequence number*, ma fornisce anche una base inziale per misurare il *Round Trip Time (RTT)* (Figura 3.4), che rappresenta il tempo totale necessario per un pacchetto di dati per viaggiare dal mittente al destinatario e ritorno. La misurazione di questo valore è essenziale per ottimizzare il controllo di flusso e la gestione della congestione della rete [7].

Ritrasmissioni

In seguito viene descritto brevemente il meccanismo di ritrasmissione utilizzato da *TCP*, che risulta necessario per comprendere alcuni concetti della tesi.

Il *TCP* è un protocollo progettato per garantire l'affidabilità nella comunicazione dei dati. Durante la trasmissione i pacchetti possono venire persi o essere danneggiati a causa di vari fattori come congestione della rete, interferenze o errori *hardware*. La ritrasmissione dei questi segmenti è uno dei meccanismi chiave per garantire l'integrità dei dati. Questo processo si basa sulla logica degli *acknowledgements*. Quando un segmento viene inviato, il mittente attende una conferma di ricezione (*ACK*) dal destinatario. Se questa conferma non viene ricevuta entro un determinato intervallo di tempo, chiamato *Retransmission TimeOut (RTO)*, il mittente assume che il segmento sia stato perso e lo ritrasmette. Il calcolo del RTO è dinamico e dipende fortemente dal *RTT* e da numerosi altri valori.

Sicurezza

Nella sua forma nativa, il protocollo *TCP*, non offre meccanismi di sicurezza come autenticazione, integrità dei dati o confidenzialità. Infatti, *TCP* non implementa alcuna funzionalità di crittografia o altri sistemi con lo scopo di proteggere i dati durante la trasmissione. Per colmare questa mancanza, vengono utlizzati protocolli aggiuntivi come il *Transport Layer Security (TLS)* e il *Secure Sockets Layer (SSL)* (suo predecessore). Questi protocolli crittografici operano al *livello di presentazione* del modello *ISO/OSI* e di conseguenza ad un livello superiore rispetto al *TCP*. Forniscono entrambi le funzionalità di sicurezza essenziali, garantendo una comunicazione protetta e affidabile.

Nel contesto di questo studio, ci si soffermerà in particolare su *TLS 1.3*, l'ultima versione di questo protocollo. In Figura 3.5 è riportata la nuova procedura di *handshake*, essenziale per negoziare i parametri di sicurezza e stabilire una comunicazione cifrata.
I passaggi coinvolti in questo processo sono i seguenti:

1. **Client invia un messaggio ClientHello al Server**: Il messaggio contiene una lista dei cifrari supportati dal client;

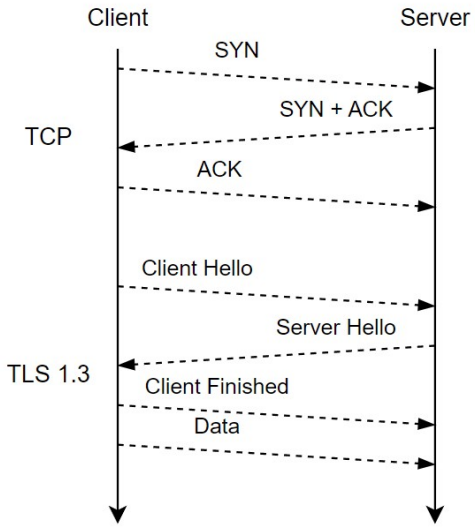


Figura 3.5: *Processo three-way handshake in TCP con TLS 1.3*

2. **Server risponde con un messaggio ServerHello al Client**: Il Server risponde con un messaggio che contiene:

   - Il cifrario selezionato tra quelli inviati dal Client;
   - La chiave pubblica del server per l'algoritmo di scambio chiavi scelto.

   Segue poi il messaggio *Certificate*, che include il certificato del server, e un messaggio *Finished* che conclude la sua parte dell'*handshake*;

3. **Client invia un messaggio Finished**: Il Client dopo aver ricevuto il messaggio *Finished* dal Server possiede tutte le informazioni necessarie per confermare il completamento dell'*handshake* e invia a sua volta un messaggio *Finished* per confermare il termine di tale operazione.

È importante notare che con il *TLS 1.3* si ha un *2RTT*, ovvero vengono richiesti due *Round Trip* prima dell'effettivo invio dei dati [8].

UDP (User Datagram Protocol)

Dopo aver esaminato il *Transmission Control Protocol (TCP)*, si procede ora all'analisi del *User Datagram Protocol (UDP)*. Diversamente dal *TCP*, questo protocollo si distingue come un protocollo senza connessione. Ciò significa che non è necessario stabilire una connessione dedicata tra *client* e *server* prima di iniziare lo scambio di dati. Questa caratteristica, unita a un *header* di soli 8 *byte* e alla mancanza di un controllo sia di Flusso che di Errore, consente a *UDP* di essere più veloce e leggero rispetto a *TCP*.
Nonostante la sua relativa semplicità, una trattazione esaustiva dell'UDP andrebbe comunque oltre gli scopi di questa tesi. Di conseguenza ci si concentrerà sulla struttura di un *datagramma UDP* descritto in Figura 3.6.

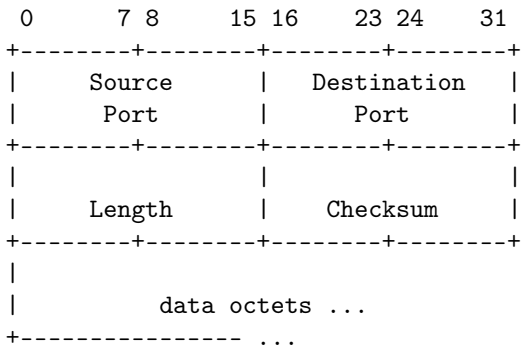


Figura 3.6: *Composizione di un Datagramma UDP*

Si può facilmente notare che l'*header* di un datagramma *UDP* è notevolmente più semplice rispetto a quello di un segmento *TCP*. In quanto questo è composto da soli quattro campi, i quali sono :

- ***Source Port - Destination Port***: Identificano rispettivamente il numero della porta di origine e destinazione. In questo caso il campo relativo alla *source port* può essere omesso ed in tal caso settato a 0.
- ***Lenght***: Questo campo specifica la lunghezza in byte dell'*header* e del *payload*. La lunghezza minima è di 8 *bytes* mentre il limite superiore è di 65,607 *bytes*.
- ***Checksum***: Questo campo è opzionale e può essere usato per effettuare dei controli di errore sul datagramma.

La semplicità di questa struttura riflette a pieno i suoi casi d'uso, dove i punti di forza principali sono la velocità e l'efficienza [9]. In particolare l'assenza di complessi meccanismi di controllo e gestione, invece presenti nel *TCP*, rende l'*UDP* particolarmente adatto per applicazioni che richiedono una trasmissione rapida e in cui la perdita di alcuni pacchetti non invalida totalmente il servizio. Ne sono un esempio i giochi online, lo streaming video e le chiamate *VOIP*[9].

3.1.3 QUIC

I protocolli di trasporto tradizionali come *TCP* e *UDP* hanno servito Internet per decenni, ma nell'era delle comunicazioni moderne, dove le comunicazioni mobili e le applicazioni ad alta velocità dominano le trasmissioni, mostrano alcuni limiti significativi. *TCP*, nonostante la sua affidabilità, soffre di latenza elevata a causa dell'*overhead* dovuto alla gestione del Flusso. Dall'altra parte, *UDP*, pur essendo veloce, manca di meccanismi che ne garantiscono sicurezza e affidabilità.
Sviluppato inizialmente da *Google* nel 2012, *Quick UDP Internet Connections (QUIC)* è stato pensato per affrontare le sfide che i protocolli tradizionali faticavano a gestire, come le performance sulle reti mobili, la sicurezza e la latenza ridotta. Offrendo una soluzione che combina la velocità dell'*UDP* alla sicurezza e affidabilità del *TCP*. Questo nuovo protocollo si è rapidamente affermato nel mondo delle comunicazioni, tanto da essere standardizzato dall'*IETF*[10] nel 2021.

[9] Voice Over IP (VOIP)
[10] Internet Engineering Task Force (IETF)

Inoltre *QUIC* è stato sviluppato per integrarsi con *HTTP*[11], chiamando *HTTP/3* la versione che utilizza *QUIC* al posto di *TCP* come protocollo di trasporto [10]

Introduciamo ora gli elementi di *QUIC* che vengono affrontati in questo studio, concentrandoci in particolare su:

1. La struttura del protocollo;
2. Il processo di *handshake*;
3. Multiplexing;
4. Sistemi di sicurezza;

Struttura del Protocollo

Il protocollo *QUIC*, come suggerisce il nome, è costruito sopra al protocollo *UDP*, ovvero tutte le informazioni necessarie al funzionamento di *QUIC* sono contenute nel payload del datagramma *UDP* (Figura 3.6). Questa scelta mira principalmente a evitare l'*ossification*[12], un problema comune nello sviluppo di nuovi protocolli. L'*ossification* si riferisce alla difficoltà di introdurre nuovi protocolli o modificare quelli esistenti a causa della presenza di dispositivi di rete che si aspettano dei comportamenti specifici dai protocolli conosciuti. Tali dispositivi, noti anche come *middle box*[13], sono gli elementi di rete in cui i pacchetti transitano. Costruendo *QUIC* sopra *UDP*, si sfrutta il fatto che l'*UDP* è ampiamente riconosciuto dalle *middle box*, riducendo il rischio che un pacchetto *QUIC* venga scartato. Questo approccio permette a *QUIC* di implementare le proprie funzionalità all'interno del payload *UDP*, mantenendo la compatibilità con l'infrastruttura di rete esistente.
Inoltre, l'utilizzo di *UDP* non introduce alcun *overhead*[14] significativo e oltre a ciò è già integrato nei vari *sistemi operativi*, cosa che non sarebbe vera se *QUIC* si basasse direttamente su *IP*. Come illutrato nella Figura 3.7, *QUIC* si basa su *UDP* in modo analogo a come *TCP* si basa su *IP*. Nonostante questa differenza strutturale, *QUIC* implementa le stesse funzionalità chiave di *TCP*, seppur in modo diverso:

- Orientato alla connessione;
- Recupero delle perdite;
- Controllo del flusso;
- Ritrasmissione;
- Controllo della congestione.

Identificatori di Connessione

A differenza di *TCP*, *QUIC* adotta un approccio diverso per l'identificazione delle connessioni. Mentre in *TCP* le connessioni sono identificate tramite la tupla:

[11] HyperText Transfer Protocol (HTTP)
[12] *Ossification*
[13] *Middle box*
[14] *Overhead*

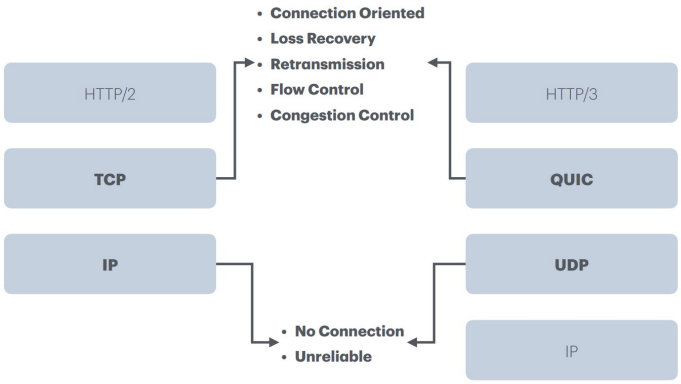


Figura 3.7: *Confronto tra gli stack di protocollo TCP e QUIC*

[Indirizzo IP sorgente, Porta sorgente] ↔ [Indirizzo IP destinazione, Porta destinazione]

QUIC introduce un concetto diverso: il *Connection ID*. Questo identificatore consente di riconoscere la connessione in maniera indipendente dagli *indirizzi IP* o dalle *porte* utilizzate. Il vantaggio principale di questo approccio è la capacità di mantenere attiva una connessione anche in caso di cambiamento dell'*indirizzo IP* o della *porta*, situazione che nelle reti moderne è molto frequente.

Un esempio concreto che illustra i vantaggi di questo approccio si può osservare nel comportamento dei dispositivi mobili durante il passaggio tra diverse reti. Con una connessione *TCP*, quando un dispositivo migra da una rete *WiFi* a una *rete mobile*, è richiesta una riconnessione completa, come si può vedere in Figura 3.8a.

Al contrario, *QUIC* gestisce questa transazione in modo più efficiente come mostrato nella Figura 3.8b. Grazie al *Connection ID*, *QUIC* riesce a mantenere la connessione attiva nonostante il cambio di rete, eliminando la necessità di effettuare una riconnessione e garantendo un servizio costante [11].
Oltre a questo vantaggio fondamentale, l'utilizzo del *Connection ID* offre ulteriori benefici:

- **Load balancing**: Grazie al *Connection ID* i server possono distribuire il carico in modo più dinamico senza avere interruzioni nelle connessioni esistenti, migliorando le prestazioni e la scalabilità [12].
- **Maggiore privacy**: Il *Connection ID* può essere cambiato frequentemente, aumentando la protezione della privacy e riducendo la tracciabilità delle connessioni [13].

Packet Number

QUIC introduce un nuovo meccanismo per la gestione dei numeri di sequenza, ogni pacchetto trasmesso in una connessione *QUIC* viene indentificato tramite un *Packet Number*, un numero univoco compreso fra 0 e 2^{62}-1. Questo numero viene utilizzato per identificare il *nonce crittografico*[15] per la protezione dei pacchetti. Per ogni *endpoint*[16]

[15] *Nonce Crittografico*
[16] *Endpoint*

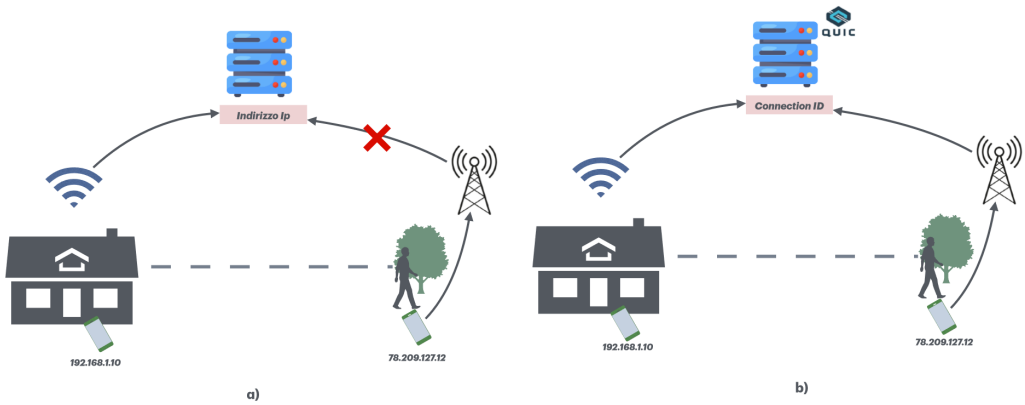


Figura 3.8: *Confronto meccanismo di hand-off nel caso di connessione TCP e QUIC*

(a) Con TCP, quando l'utente passa da rete WiFi a rete mobile, ciò comporta un cambiamento del suo indirizzo IP. Tale cambio causa la terminazione della connessione esistente. Poiché TCP identifica la connessione tramite l'indirizzo IP.

(b) Con QUIC, quando l'utente passa da WiFi a rete mobile cambiando il suo indirizzo IP, la connessione non viene terminata. Questo perché QUIC utilizza il Connection ID invece dell'indirizzo IP per identificare la connessione, permettendo di mantenere la sessione attiva nonostante il cambio di rete.

il *Packet Number* è diverso per il traffico in entrata e in uscita. Tale dettaglio permette a ciascun *endpoint* di gestire i propri flussi di dati in maniera indipendente, permettendo maggiore flessibilità e sicurezza. *QUIC*, inoltre, suddivide il *Packet Number* in tre spazi distinti:

i *Initial Space*, contiene tutti i pacchetti iniziali.

ii *Handshake Space*, contiene tutti i pacchetti di *handshake*.

iii *Application Data Space*, contiene tutti i pacchetti *0-RTT* e *1-RTT*

Concettualmente, un *packet number space* è il contesto nel quale un pacchetto viene processato e confermato. Questo garantisce che i pacchetti nei *number space* diversi siano crittograficamente separati.
Ogni spazio inizia con un *Packet Number* pari a 0, ogni pacchetto successivo incrementa il numero di 1 alla volta [14].

Formato dei Pacchetti

Un datagramma *UDP* può contenere uno o più pacchetti *QUIC*. Ciascun pacchetto *QUIC* è composto da un *header* e da *N frame*, come illustrato nella Figura 3.9 secondo le specifiche definite nel RFC 9000 [14].
Gli *header* si distinguono in due categorie, *long header* e *short header*.

Il *long header* viene utilizzato principalmente durante la fase di inizializzazione della connessione. Di seguito, viene riportata una breve descrizione dei campi presenti (Figura 3.10) [14].

- ***Header Form***: 1 bit, indica il tipo di *header*, dove 1 indica *long header*.

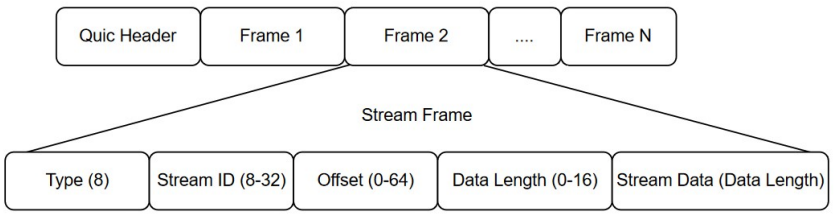


Figura 3.9: *Composizione di un pacchetto QUIC*

```
Long Header Packet {
  Header Form (1) = 1,
  Fixed Bit (1) = 1,
  Long Packet Type (2),
  Type-Specific Bits (4),
  Version (32),
  Destination Connection ID Length (8),
  Destination Connection ID (0..160),
  Source Connection ID Length (8),
  Source Connection ID (0..160),
  Type-Specific Payload (..),
}
```

Figura 3.10: *Composizione di un Long Header QUIC*

Type	Name
0x00	*Initial*
0x01	*0-RTT*
0x02	*Handshake*
0x03	*Retry*

Figura 3.11: *Tipi di Long Header QUIC*

- ***Fixed Bit***: 1 bit, il valore fissato è 1 a meno che non sia un pacchetto di *Negoziazione di Versione*. I pacchetti che contengono 0 in questo campo devono venire scartati.
- ***Long Packet Type***: 2 bit, specifica il tipo di pacchetto. I tipi di pacchetti sono specificati nella Figura 3.11
- ***Type-Specific Bits***: 4 bit, contiene bit specifici per il tipo di pacchetto.
- ***Version***: 32 bit, indica la versione di *QUIC* in uso.
- ***Destination Connection ID Length***: 8 bit, indica la lunghezza del *Connection ID* di destinazione.
- ***Destination Connection ID***: 0 a 160 bit, indica il *Connection ID* di destinazione.
- ***Source Connection ID Length***: 8 bit, indica la lunghezza del *Connection ID* di origine.
- ***Source Connection ID***: 0 a 160 bit, indica il *Connection ID* di origine.
- ***Type-Specific Payload***: lunghezza variabile, contiene il payload specifico per il tipo di pacchetto.

Una discussione approfondita sui singoli tipi di pacchetto presenti in Figura 3.11 sarebbe poco inerente motivo per cui si descriverà il ruolo di ogni tipo senza entrare nel dettaglio.

Initial Packet ha il compito di trasportare il primo *Crypto Frame* mandato dal *client* e dal *server* per fare lo scambio delle chiavi di crittografia, e si occupa anche di trasportare l'*acknowledge* in entrambe le direzioni.

0-RTT ha il compito di trasportare gli *early data* (dati anticipati) che il *client* invia al *server* come parte della prima comunicazione, questo permette di ridurre la latenza. Si osserverà meglio questo nella sezione dedicata all'*handshake*.

Handshake Packet ha il compito di trasportare i messaggi crittografici necessari per il processo di handshake. Una volta che il *client* ha ricevuto un *Handshake packet* dal *server*, il *client* utilizza pacchetti di *handshake* per inviare i successivi messaggi criptografici di *handshake* e riconoscimenti al *server*.

Retry Packet ha il compito di trasportare un *token* di validazione creato dal *server*. Viene usato dal *server* quando vuole effettuare un *retry*, ovvero richiedere al *client* di ripetere l'invio di una richiesta iniziale.

Passiamo ora ad analizzare il *short header*, utilizzato esclusivamente dopo che le chiavi sono state negoziate. Esiste un'unica variante di questo *header*, noto come *RTT-1* (illustrato nella Figura 3.12) [14].

```
1-RTT Packet {
    Header Form (1) = 0,
    Fixed Bit (1) = 1,
    Spin Bit (1),
    Reserved Bits (2),
    Key Phase (1),
    Packet Number Length (2),
    Destination Connection ID (0..160),
    Packet Number (8..32),
    Packet Payload (8..),
  }
```

Figura 3.12: *Composizione di un Short Header QUIC*

- ***Header Form***: 1 bit, indica il tipo di *header*, dove 0 indica *short header*.
- ***Fixed Bit***: 1 bit, il valore fissato è 1. I pacchetti che contengono 0 in questo campo devono venire scartati.
- ***Spin Bit***: 1 bit, viene utilizzato per il monitoraggio passivo della latenza, il *server* riflette il valore dello *spin bit* ricevuto mentre il *client* lo inverte dopo ogni *RTT*.
- ***Reserved Bits***: 2 bit, sono dei bit riservati e protetti usando l'*header protection*, il loro valore dopo aver rimosso la protezione deve essere 0, in caso contrario viene segnalato un errore di connessione.
- ***Key Phase***: 1 bit, indica la fase della chiave, ovvero permette a chi riceve il pacchetto di identificare le chiavi usate per proteggere il pacchetto. Anche questo campo è protetto usando l'*header protection*.
- ***Packet Number Lenght***: 2 bit, indica la lunghezza effettiva del campo *Packer Number*
- ***Destination Connection ID***: 0 a 160 bit, indica il *Connection ID* scelto da chi riceve il pacchetto.

- ***Packet Number***: 8 a 32 bit, contiene il *Packet Number*.
- ***Packet Payload***: 8 a lunghezza variabile, contiene il *payload* protetto.

Handshake

Procediamo ora con l'analisi del processo di *handshake* in *QUIC* (Figura 3.13), evidenziando le differenze rispetto al *handshake* del protocollo *TCP* (Figura 3.13a).

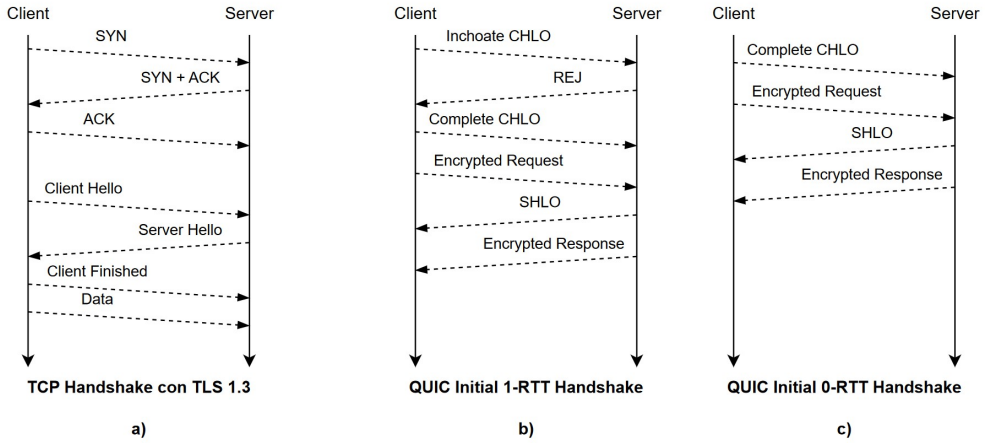


Figura 3.13: *Confronto processi di handshake QUIC e TCP*

(a) Processo di handshake TCP con TLS 1.3.

(b) QUIC First-time Connection Establishment.

(c) QUIC 0-RTT Connection Establishment.

QUIC permette di ridurre significativamente il numero di scambi necessari per l'inizializzazione della connessione rispetto a *TCP*. In particolare, si possono identificare due scenari:

1. ***First-time Connection Establishment***: Avviene quando il *client* si connette per la prima volta al *server*. Viene inviato un messaggio di *Inchoate Client Hello (CHLO)* a cui il *server* risponde con un messaggio di *Reject (REJ)*. Questo messaggio contiene :

   i La configurazione del server, incluso il valore pubblico di *Diffie-Hellman*[17] a lungo termine del server,

   ii Una catena di certificati che autenticano il *server*.

   iii Una firma delle configurazioni del *server* usando una chiave privata presa dal certificato finale della catena.

   iv Un *Source Address Token*, ovvero un blocco di crittografia non autenticata che contiene l'*indirizzo IP* pubblico del *client*.

   Questo token viene utilizzato per verificare l'identità del *client* nelle comunicazioni future. Il *client* usa queste informazioni per autenticare la configurazione del

[17] *Diffie-Hellman*

server e invia un messaggio *CHLO* completo (*finished*), contenente il valore *Diffie-Hellman* temporaneo del *client* (Figura 3.13b) [1].

2. ***0-RTT Connection Establishment***: Dopo il primo handshake, il client possiede già le chiavi iniziali per la connessione. Quindi procede con l'invio del *CHLO* completo e può iniziare a inviare dati applicativi al *server*. Se l'*handshake* ha successo viene restituito un messaggio *Server Hello (SHLO)* cifrato con le chiavi iniziali e contenente il nuovo valore temporaneo di *Diffie-Hellman* (Figura 3.13c) [1] .

A differenza di *TCP*, che richiede 2 *round trip* per l'invio dei dati dopo l'inizializzazione della connessione, *QUIC* nel caso migliore riesce a ridurre questo numero a 0.

Multiplexing

Il *multiplexing* è una tecnica utilizzata nei sistemi di comunicazione per trasmettere più flussi di dati attraverso un singolo canale di comunicazione. Tale meccanismo consente di ottimizzare l'uso delle risorse, permettendo a più flussi di dati di condividere la stessa connessione.
In *QUIC*, il *multiplexing* è gestito a livello di *stream*[18], permettendo a ogni connessione di ospitare un numero potenzialmente illimitato di *stream* indipendenti.
Ogni *stream* (flusso di dati) è una sequenza ordinata e affidabile di dati che coesiste in parallelo all'interno della stessa connessione. Grazie a questo, la perdita di pacchetti in uno *stream* non interrompe il flusso degli altri, permettendo un utlizzo più efficiente della connessione e migliorando notevolemente le prestazioni in termini di latenza e throughput. Ciò è in contrasto con *TCP*, in cui viene gestita una singola sequenza di dati per connessione.
Un esempio nel quale questo meccanismo risulta particolarmente utile è nelle applicazioni web moderne, dove spesso vengono richiesti simultaneamente più componenti.

Questo approccio elimina il problema del *Head of Line Blocking (HoL)*, un fenomeno comune nel *TCP* dove la perdita o il ritardo di un pacchetto obbliga l'intero flusso di dati a interrompersi finché il pacchetto non viene ritrasmesso e ricevuto correttamente.

Sicurezza del Protocollo

Uno degli aspetti fondamentali alla base di *QUIC* è l'attenzione alla sicurezza. Fin dalla sua progettazione iniziale, è stato concepito per migliorare l'affidabilità e la sicurezza delle connessioni. Sono diversi i meccanismi integrati all'interno di *QUIC* per ottenere questo obbiettivo.

In *QUIC* l'integrazione della sicurezza tramite *TLS 1.3* è parte integrante del protocollo. Il processo di stabilimento della connessione incorpora nello stesso *handshake* la negozazione e l'instaurazione della sicurezza *TLS*. Al contrario di *TCP*, dove *TLS* è un protocollo aggiuntivo che viene usato per aggiungere un livello di sicurezza sopra alla connessione *TCP*. Questo comporta molteplici differenze e vantaggi.

L'integrazione delle fasi necessarie al *TLS* nel processo di stabilimento della connessione *QUIC* permette di ridurre il numero di round trip necessari. Inoltre, con *QUIC*, non solo gli *user data* vengono crittografati, ma l'intero pacchetto (*header*

[18] *Stream*

escluso) [11].

Un ulteriore aspetto fondamentale nella sicurezza del protocollo è rappresentato dalla crittografia *end-to-end*[19], ciò assicura che i dati trasmessi tra il *client* e il *server* siano protetti da intercettaazioni o manipolazioni da parte di terzi. Questo avviene tramite l'uso di algoritmi crittografici come *Diffie-Hellman* nella prima fase e successivamente una chiave simmetrica condivisa. Inoltre viene utilizzato un rinnovamento periodico delle chiavi per garantire una maggiore sicurezza e una protezione continua [13].

Un altro meccanismo impiegato per garantire l'integrità dei dati è l'utilizzo di un *checksum*, che consente di verificare che i dati non siano stati alterati durante la trasmissione. Se il destinatario riscotra discrepanze tra il valore del campo *checksum* e i dati stessi, questi verranno rifiutati e scartati. Inoltre, *QUIC* include nei pacchetti un *hash* crittografico dei dati, che il destinatario può ricalcolare per verificare la presenza di eventuali corruzioni nei dati [13].

Ritrasmissioni

In seguito viene descritto brevemente il meccanismo di ritrasmissione utilizzato da *QUIC*, che risulta necessario per comprendere alcuni concetti della tesi.

In particolare, *QUIC* fa uso di un *Probe Timeout (PTO)* a differenza del *TCP* che invece utilizza il *RTO* tradizionale. Il *PTO* è un meccanismo progettato per essere più reattivo e flessibile nella gestione delle perdite di pacchetti e nei ritardi. Questo meccanismo consente di ritrasmettere i pacchetti con un tempo di attesa significativamente ridotto rispetto all'*RTO* tradizionale. A differenza di quest'ultimo, che aumenta il timeout in modo esponenziale con ogni ritrasmissione fallita, il *PTO* mantiene un intervallo di attesa limitato da una costante massima, migliorando così la reattività e riducendo il tempo per recuperare i pacchetti persi [15].

Viene inoltre usato per gestire separamente i *packet number space* dando priorità ad una fase rispetto ad un'altra. Questo garantisce a *QUIC* una maggiore efficienza in reti con alta latenza e perdite di pacchetti.

3.1.4 MPTCP

Multipath TCP (MPTCP) rappresenta un'evoluzione del protocollo *TCP* tradizionale. Si tratta di un insieme di estensioni progettate con lo scopo di ampliare le funzionalità del *TCP* standard, introducendo il concetto di *multipath*[20]. Questa innovazione consente a una singola connessione di operare simultaneamente su molteplici percorsi di rete.

L'idea alla base di *MPTCP* è quella di sfruttare al massimo le varie risorse di rete disponibili, permettendo a una connessione di utlizzare in contemporanea diverse interfacce o percorsi.

Questa sezione si focalizzerà primariamente sul funzionamento di *MPTCP*, analizzando

[19] *End-to-End*
[20] *Multipath*

i suoi principi base, la struttura del protocollo e i vantaggi che offre rispetto al *TCP* tradizionale.

Struttura del Protocollo

MPTCP introduce una struttura innovativa al livello di trasporto, suddividendolo in due sottolivelli distinti, come illustrato nella Figura 3.14.

Il sottolivello superiore di *MPTCP* opera end-to-end ed è responsabile della gestione complessiva dei vari *subflows* sottostanti. Per fare questo, implementa le seguenti funzioni:

i *Path Management*: Gestisce la scoperta, l'aggiunta e la rimozione di percorsi di rete disponibili.

ii *Packet Scheduling*: Politica con cui si decide in che modo distribuire i pacchetti tra i vari *subflow* attivi.

iii *Congestion Control*: Implementa un controllo della congestione a livello *MPTCP*, unendo i meccanismi di controllo della congestione dei singoli *subflow*.

iv *Subflow Interface*: Si occupa di mappare i dati dell'applicativo sui *subflow* appropriati, gestendo la riordinazione e presentando un flusso di dati coerente all'applicazione.

Il sottolivello inferiore, è responsabile dei singoli *subflow*. Ogni *subflow* viene trattato come un flusso *TCP* indipendente. Questa struttura permette a *MPTCP* di essere *retro-compatibile* con le reti esistenti, evitando problemi come l'*ossification*. Inoltre, consente al componente *TCP* di operare segmento per segmento, garantendo le funzionalità essenziali del *TCP* tradizionale.

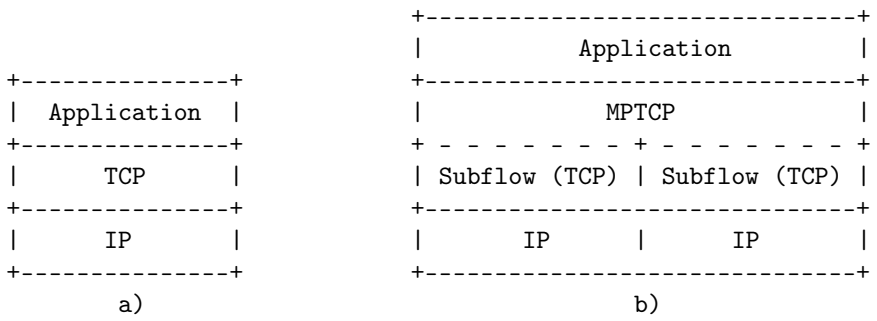


Figura 3.14: *Confronto tra gli stack di protocollo TCP e MPTCP standard*

(a) Nel TCP, il protocollo opera su un singolo flusso di dati, utilizzando un unico percorso di rete tra il livello applicazione e lo strato IP.

(b) Con MPTCP, il protocollo gestisce più sottoflussi TCP simultaneamente, permettendo l'utilizzo di percorsi di rete multipli tra il livello applicazione e gli strati IP sottostanti.

Un aspetto fondamentale di questa architettura è la sua trasparenza. Dal punto di vista applicativo, *MPTCP* risulta una singola connessione *TCP* standard, nascondendo la complessità della gestione *multipath* [16].

Formato dei Pacchetti

Essendo un'estensione del *TCP* tutte le operazioni del *MPTCP* sono contenute all'interno dei campi opzionali dell'*header TCP*. Per identificare le opzioni specifiche di *MPTCP*, il campo *Kind* nell'*header TCP* è impostato al valore 30, come assegnato dalla *Internet Assigned Number Authority (IANA)*.

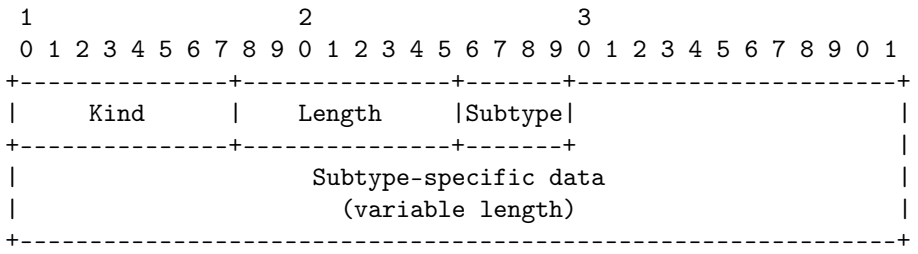


Figura 3.15: *Composizione opzione MPTCP*

Il campo *Subtype* specifica la funzione dell'opzione *MPTCP*. La Tabella 3.1 presenta un elenco completo dei sottotipi disponibili come definito nel RFC 8684 [17]. Di seguito, viene riportata una breve descrizione di ogni tipo.

Value	Symbol	Name
0x0	*MP_CAPABLE*	Multipath Capable
0x1	*MP_JOIN*	Join Connection
0x2	*DSS*	Data Sequence Signal
0x3	*ADD_ADDR*	Add Address
0x4	*REMOVE_ADDR*	Remove Address
0x5	*MP_PRIO*	Change Subflow Priority
0x6	*MP_FAIL*	Fallback
0x7	*MP_FASTCLOSE*	Fast Close
0x8	*MP_TCPRST*	Subflow Reset

Tabella 3.1: *Tabella dei sottotipi dell'opzione MPTCP*

Multipath Capable viene utilizzato durante il processo di avvio di una connessione, verifica che entrambi gli *endpoint* supportino *MPTCP* e contiene ulteriori informazioni per l'autenticazione di ulteriori *subflow*.

Join Connection viene utilizzato per stabilire nuovi *subflow* all'interno di una connessione *MPTCP* esistente, utilizza le informazioni condivise nel *Multipath Capable* per autenticare il *subflow* come parte della connessione principale.

Data Sequence Signal gestisce la mappatura dei dati del livello applicativo sui *subflow*, gestendo la riordinazione e la ritrasmissione dei segmenti. Utilizza un *Data Sequence Number* a 64 bit per numerare in modo univoco tutti i dati trasmessi attraverso la connessione *MPTCP*.

Add Address permette a un *endpoint* di informare la disponibilità di un *indirizzo IP* aggiuntivo disponibile per la connessione, senza stabilire un *subflow*.

Remove Address consente a un *endpoint* di notificare che un *indirizzo IP* precedentemente annunciato non è più disponibile.

Change Subflow Priority consente di modificare la priorità di un *subflow*, in-

fluenzando le decisioni di *Scheduling* del traffico.

Fallback segnala la presenza di un problema che richiede il *fallback* al *TCP* normale.

Fast Close viene utilizzato per segnalare ad un *endpoint* che la connessione verrà chiusa.

Subflow Reset consente di resettare un singolo *subflow* senza influenzare l'intera connessione *MPTCP*.

In conclusione, *MPTCP*, essendo un insieme di estensioni del protocollo *TCP*, ne eredita molte caratteristiche di sicurezza. Questa continuità, offre numerosi vantaggi in termini di compatibilità e robustezza, ma comporta anche alcune problematiche.

Da un lato, MPTCP utilizza tecniche di sicurezza consolidate come il *three-way-handshake* (opportunamente modificato per gestire il *multipath*) e i meccanismi per il controllo della gestione.

Dall'altra parte, questa stretta dipendenza con *TCP* implica che *MPTCP* sia soggetto a molte delle stesse problematiche e vulnerabilità del *TCP*. Inoltre la natura *multipath* del protocollo introduce una nuova possibile superfice d'attacco, richiedendo l'implementazione di meccanismi specifici per autenticare e garantire l'integrità dei singoli *subflow*.

3.2 Problematiche Attuali

Nel contesto delle reti moderne le comunicazioni su reti mobili stanno assumendo un'importanza sempre maggiore, con un numero crescente di dispositivi connessi e un volume di dati in crescita continua. La diffusione di smartphone e dispositivi *IoT* ha alimentato un incremento esponenziale del traffico dati, rendendo le reti mobili fondamentali per la la vita quotidiana e per molteplici attività economiche. In questo contesto, garantire l'efficienza, la sicurezza e la correttezza del traffico dati su queste reti è diventato sempre più una priorità.

Un aspetto critico delle reti mobili è la tariffazione del traffico, che spesso si basa sul volume consumato dall'utente. Meccanismi di rete come la ritrasmissione dei pacchetti persi possono influenzare questo valore, rendendo complesso il calcolo del consumo. Questo comporta che i diversi operatori del traffico cellulare devono decidere che politica adottare per i pacchetti ritrasmessi. Un primo approccio è quello di addebitare ogni pacchetto per l'utilizzo dell'architettura delle reti mobili. Tuttavia, questa politica espone gli utenti a un possibile *Usage-Inflation Attack*, dove un avversario potrebbe aumentare artificialmente il traffico dati ritrasmettendo pacchetti. Un secondo approccio è quello di escludere i pacchetti ritrasmessi dalla contabilizzazione. Questo scenario, nonostante eviti lo *Usage-Inflation Attack*, aumenta notevolemente la complessità dell'implementazione, richiedendo metodi per il monitoraggio accurato del traffico. Inoltre, potrebbe essere vulnerabile a *free-riding Attack*, dove gli attaccanti mascherano il proprio traffico all'interno di finti pacchetti ritrasmessi, evadendo così la tariffazione [6].

Il problema specifico affrontato in questo studio riguarda lo scenario presentato nel

primo approccio. L'obiettivo principale della ricerca è stato quello di esplorare eventuali vulnerabilità di *QUIC* che potrebbero essere sfruttate per manipolare i sistemi di contabilizzazione del traffico mobile. In particolare, lo studio si è concentrato sulle possibili problematiche che causano un aumento della ritrasmissione o eventuali metodi per forzare ritrasmissioni, con lo scopo di aumentare artificialmente la contabilizzazione del traffico.

Capitolo 4

Processi e Metodi

Questo capitolo fornisce in dettaglio l'ambiente di ricerca utilizzato, le tecnologie impiegate e descrive gli esperimenti condotti. Provvede a dare inoltre tutte le informazioni necessarie per replicare gli esperimenti.

4.1 Ambiente

Questa sezione offre una panoramica completa dell'ambiente di lavoro e delle tecnologie impiegate nello sviluppo degli esperimenti descritti successivamente. Di seguito, viene riportata una descrizione degli strumenti utilizzati durante lo svolgimento del progetto (riassunte con le relative versioni nella Tabella 4.2).

La totalità del progetto è stata svolta su ***Ubuntu***[1]. Questa scelta di sistema operativo è dovuta al vasto supporto di strumenti per l'analisi e lo sviluppo.

Per l'analisi del traffico di rete, è stato utilizzato ***Wireshark***[2], uno strumento molto diffuso nel panorama del traffico di rete e che ha permesso di esaminare nel dettaglio il comportamento dei protocolli oggetti dello studio.

La sperimentazione ha coinvolto l'uso di diversi *browser web*, in particolare ***Google Chrome***[3] e ***Firefox***[4]. Questi due applicativi sono stati impiegati per testare e confrontare l'implementazione dei protocolli in diverse situazioni.

L'ambiente di test è stato realizzato tramite **Oracle VM VirtualBox**[5], un diffuso ambiente di virtualizzazione, che consente di creare e gestire più macchine virtuali.

Per la condivisione e mantenimento del codice si è usato **GitHub**[6] e **Git**. In particolare si è rivelato vantaggioso in quanto ha permesso di lavorare efficientemente con

[1] https://ubuntu.com/
[2] https://www.wireshark.org/
[3] https://www.google.com/intl/it_it/chrome/
[4] https://www.mozilla.org/it/firefox/
[5] https://www.virtualbox.org/
[6] https://github.com/

fork[7] di librerie pubbliche, consentendo una gestione dinamica delle modifiche e degli aggiornamenti del codice.

4.1.1 Tecnologie Specifiche per QUIC

Per lo studio si è utilizzato **Quic-go** [18], un'implementazione sviluppata in *Go*[8] del protocollo *QUIC*. *Quic-go* è un progetto *open source* su *GitHub* che aderisce rigorosamente alle specifiche del protocollo *QUIC* definite negli *RFC*[9] 9000, 9001 e 9002 [14, 19, 20].

Quic-go non è l'unica versione disponibile del protocollo *QUIC*. Come illustrato nella Tabella 4.1, esistono numerose altre implementazioni, sia *open source* che proprietarie, ciascuna cerca di rispettare rigorosamente le specifiche definite negli *RFC*.

La scelta di utilizzare questa specifica implementazione di *QUIC* è motivata dal suo utilizzo all'interno di **Caddy**[10], un *web server* moderno e performante. *Caddy* integra *Quic-go* per offrire il supporto nativo del protocollo *QUIC* e *HTTP/3*. Inoltre, si è utilizzato **xCaddy**, un *tool* che consente di creare build personalizzate di *Caddy*, adattandolo specificatamente alle esigenze del progetto.

4.1.2 Tecnologie Specifiche per MPTCP

Per quanto riguarda *MPTCP* si è fatto riferimento alla documentazione ufficiale [21] e si è usato *Go* per sviluppare un *web server* che implementa un *socket MPTCP*.

Per garantire che le applicazioni utilizzassero *MPTCP* si è usato **mptcpize**[11], un tool specifico che forza la creazione di *socket MPTCP* al posto di quelli *TCP*.

4.2 Esperimenti

La sezione corrente esamina in dettaglio gli esperimenti condotti nel corso dello studio, illustrando la logica sottostante e le procedure impiegate per la loro realizzazione. L'obiettivo è fornire una panoramica completa delle attività sperimentali, cosicchè ci sia una maggiore comprensione sia dei metodi utilizzati che degli scopi.

Gli esperimenti sono stati creati con l'obiettivo di identificare e testare diversi metodi per aumentare il traffico dati e le ritrasmissioni. Tutte le prove sono state condotte in un ambiente locale, simulando possibili scenari di rete tramite la creazione di specifici attori.

I relativi risultati vengono analizzati nel Capitolo 5 mentre le conclusioni raggiunte sono discusse nel Capitolo 6.

[7] *Fork*
[8] *Go*
[9] Request for Comments (RFC)
[10] `https://caddyserver.com/`
[11] `https://manpages.ubuntu.com/manpages/lunar/man8/mptcpize.8.html`

Nome	Linguaggio	Licenza
Chromium	C++	BSD License
MsQuic	C	MIT License
QUIC Library (mvfst)	C++	MIT License
LiteSpeed QUIC Library (lsquic)	C	MIT License
ngtcp2	C	MIT License
Quiche	Rust	BSD-2-Clause License
quicly	C	MIT License
quic-go	**Go**	**MIT License**
Quinn	Rust	Apache License 2.0
Neqo	Rust	Apache License 2.0
aioquic	Python	BSD-3-Clause License
picoquic	C	BSD-3-Clause License
pquic	C	MIT License
QUANT	C	BSD-2-Clause License
quic	Haskell	BSD-3-Clause License
netty-incubator-codec-quic	Java	Apache License 2.0
nodejs-quic	NodeJs	MIT License
s2n-quic	Rust	Apache License 2.0
swift-quic	Swift	Apache License 2.0
TQUIC	Rust	Apache License 2.0
nginx	C	BSD-2-Clause License
HAProxy	C	GNU General Public License version 2
kwik	Java	GNU Lesser General Public License version 3

Tabella 4.1: *Tabella implementazioni QUIC*

Tipo	Nome	Versione
Applicativo	*Google Chrome*	126.0.6478.126
Applicativo	*Firefox*	127.0.2
Applicativo	*Github*	
Applicativo	*Git*	2.43.0
Applicativo	*Oracle VM VirtualBox*	7.0.20
Applicativo	*Wireshark*	4.2.6
Applicativo	*Caddy*	2.8.0
Applicativo	*xCaddy*	0.4.2
Applicativo	*mptcpize*	0.12
Modulo	*quic-go*	0.43.1
Sistema Operativo	*Ubuntu*	24.04 LTS
Linguaggio	*Go*	1.21

Tabella 4.2: *Tabella riassuntiva tecnologie usate*

4.2.1 Esperimenti QUIC

Gli esperimenti relativi al protocollo *QUIC* si articolano in due principali categorie. La prima, che comprende il primo e secondo esperimento, avviene in uno scenario in cui uno degli attori della connessione, specificamente il server, assume un comportamento

malevolo. Il secondo si svolge in una situazione in cui l'attaccante non controlla direttamente nessuna delle due parti coinvolte nella comunicazione, ma si suppone sia in grado di monitorare e manipolare il traffico di rete.

Per la realizzazione degli esperimenti si è utilizzato un *fork* appositamente modificato del progetto *Quic-go* [22], con cui è stato possibile implementare il *server* malevolo e condurre i vari test. La simulazione degli attori dello scenario è stata eseguita utilizzando macchine virtuali con all'interno *Ubuntu*. Per quanto riguarda il *client*, si sono utilizzati i *browser google-chrome* e *firefox* per stabilire la connessione alla *pagina web*. Sul lato *server*, è stato impiegato *Caddy* come *web server*.

Esperimento 1: Server Web QUIC che ignora gli ACK

L'idea alla base di questo esperimento è simulare un *server web Quic* con un comportamento non convenzionale. Il *server* è configurato per operare come se non ricevesse mai conferme *(acknowledge)* per i pacchetti inviati, mantenendo al contempo un *PTO* impostato a zero. Di conseguenza, il *server* continua a ritrasmettere i pacchetti, presumendo erroneamente che non siano mai giunti a destinazione. Con il *PTO* azzerato, queste ritrasmissioni avvengono in rapida successione, quasi simultaneamente.

Per implementare queste modifiche, si é utilizzato il seguente approccio :

i **Modifiche del codice sorgente di *quic-go***

- Si è modificata la gestione degli *ACK* e la logica che gestisce il *PTO*.

ii **Creazione di una build personalizzata di *Caddy***

- Si è utilizzato *xCaddy* per la creazione di una versione modificata del *server web Caddy* che usa il mio *fork* di *quic-go* con le modifiche apportate.

In particolare, le modifiche operano dopo la fase di *handshake*, così da garantire l'inizializzazione della connessione. Successivamente, il *server* agirà come modificato, continuando a ritrasmettere i vari dati.

In questo scenario, ci si aspetta che il *client* riconosca le ritrasmissioni e le segnali al *server*. Eventuali segnali e richieste di chiusura della connessione sono per questo ignorate dal *server*. Data la natura *end-to-end* delle connessioni *QUIC*, eventuali nodi intermedi nella rete non sono in grado di valutare l'integrità o la correttezza di queste ritrasmissioni. Ciò implica che solo il *client* abbia la capacità di esaminare i pacchetti e scartarli e che quindi essi vengano contabilizzati nel suo traffico dati.

Di seguito, viene riportata una descrizione di alcune delle modifiche effettuate al codice con le rispettive motivazioni.

```
// internal/ackhandler/sent_packet_handler.go
const (
        // Maximum reordering in time space before time based loss detection
            considers a packet lost.
        // Specified as an RTT multiplier.
        timeThreshold = 9.0 / 8
        // Maximum reordering in packets before packet threshold loss detection
            considers a packet lost.
        packetThreshold = 3
        // Before validating the client's address, the server won't send more than
            3x bytes than it received.
```

```
        amplificationFactor = 3
        // We use Retry packets to derive an RTT estimate. Make sure we don't set
            the RTT to a super low value yet.
        minRTTAfterRetry = 0 * time.Millisecond
        // The PTO duration uses exponential backoff, but is truncated to a maximum
             value, as allowed by RFC 8961, section 4.4.
        maxPTODuration = 0 * time.Second
)

func (h *sentPacketHandler) getScaledPTO(includeMaxAckDelay bool) time.Duration {
        // pto := h.rttStats.PTO(includeMaxAckDelay) << h.ptoCount
        pto := h.rttStats.PTO(includeMaxAckDelay)
        if pto > maxPTODuration || pto <= 0 {
                return maxPTODuration
        }
        return 0
        // return pto
}

// internal/utils/rtt_stats.go
func (r *RTTStats) PTO(includeMaxAckDelay bool) time.Duration {
        // if r.SmoothedRTT() == 0 {
        //      return 2 * defaultInitialRTT
        // }
        // pto := r.SmoothedRTT() + max(4*r.MeanDeviation(), protocol.
            TimerGranularity)
        // if includeMaxAckDelay {
        //      pto += r.MaxAckDelay()
        // }
        // return pto

        return 0
}
```

Il codice precedente elenca alcune modifiche effettuate per garantire che il *PTO* rimanga costante a 0. In particolare, sono state alterate le costanti :

- **maxPTODuration** impostato a 0 e indica il massimo valore che il *PTO* può assumere.
- **minRTTAfterRetry** impostato a 0 e indica il valore minimo impostato dopo i pacchetti di *retry*.

inoltre le funzioni che ricalcolano il *PTO* sono state cambiate così che non lo modifichino.
Di seguito, viene riportato il codice che impedisce al *server* di riconoscere gli *ACK* dei pacchetti inviati. Per ottenere questo risultato si è rimossa totalmente la logica che gestisce il ricevimento di un *acknowledge* dalla funzione responsabile di questo e nel contempo ritorna sempre una tupla *false/nil*, segnalando che nessun pacchetto è stato confermato.

```
// internal/ackhandler/sent_packet_handler.go
func (h *sentPacketHandler) ReceivedAck(ack *wire.AckFrame, encLevel protocol.
    EncryptionLevel, rcvTime time.Time) (bool /* contained 1-RTT packet */, error)
     {
    pnSpace := h.getPacketNumberSpace(encLevel)
    largestAcked := ack.LargestAcked()
    if largestAcked > pnSpace.largestSent {
        fmt.Println("received ACK for an unsent packet")
        return false, &qerr.TransportError{
            ErrorCode:    qerr.ProtocolViolation,
            ErrorMessage: "received ACK for an unsent packet",
        }
    }
    // Servers complete address validation when a protected packet is received.
    if h.perspective == protocol.PerspectiveClient && !h.
        peerCompletedAddressValidation &&
```

```
        (encLevel == protocol.EncryptionHandshake || encLevel == protocol.
            Encryption1RTT) {
        h.peerCompletedAddressValidation = true
        h.logger.Debugf("Peer doesn't await address validation any longer.")
        // Make sure that the timer is reset, even if this ACK doesn't acknowledge
            any (ack-eliciting) packets.
        h.setLossDetectionTimer()
    }
    pnSpace.largestAcked = max(pnSpace.largestAcked, largestAcked)
    // Reset the pto_count unless the client is unsure if the server has validated
        the client's address.
        if h.peerCompletedAddressValidation {
                if h.tracer != nil && h.tracer.UpdatedPTOCount != nil && h.ptoCount
                    != 0 {
                    h.tracer.UpdatedPTOCount(0)
                }
                h.ptoCount = 0
        }
        h.numProbesToSend = 0
        if h.tracer != nil && h.tracer.UpdatedMetrics != nil {
                h.tracer.UpdatedMetrics(h.rttStats, h.congestion.
                    GetCongestionWindow(), h.bytesInFlight, h.packetsInFlight())
        }
        h.setLossDetectionTimer()
        return false, nil
}
```

Ulteriori modifiche secondarie apportate, come la rimozione delle richieste di chiusura della comunicazione, sono visibili nel codice completo dell'esperimento [12] [22].

Partendo dall'idea di base di questo esperimento, sono state sviluppate diverse varianti ottenute modificando alcuni valori e parametri chiave. La lista completa è presente nell'Appendice. Le più significative di esse sono catalogate nella Tabella 4.3.

Nome	Connection Close	ACK Ignored
retransmission - 1	Ignored	All
retransmission - 2	Not Ignored	All
retransmission - 4	Not Ignored	1/2
retransmission - 5	Ignored	1/2

Tabella 4.3: *Tabella varianti esperimento 1*

Esperimento 2: Server Web con Iniezione di Pacchetti in Background

L'idea alla base di questo secondo esperimento è quella di simulare un *server web QUIC* che, pur mantenendo un comportamento apparentemente normale durante la connessione, inietta in background pacchetti aggiuntivi non richiesti.

In questo scenario, il *server* inizializza e mantiene una connessione *QUIC* standard con il *client*, agendo normalmente. Tuttavia, in parallelo a questo, il *server* è stato modificato appositamente per inviare pacchetti aggiuntivi non correlati alla comunicazione.

Anche in questo esperimento si è utilizzato *xCaddy* per la creazione di una versione modificata del *server web Caddy* con le modifiche necessarie per iniettare silenziosamente i pacchetti aggiuntivi.

[12] https://github.com/GiovanniMenon/quic-go/tree/retransmission-1

L'operazione di iniezione avviene solo dopo la fase di *handshake*, questa scelta è stata fatta per evitare potenziali interferenze, rallentamenti o errori nella prima fase della connessione.

Come nell'esperimento precedente, la natura *end-to-end* della connessione *QUIC* gioca un ruolo fondamentale. Si prevede che solo il *client* sia in grado di verificare l'autenticità dei pacchetti ricevuti. Di conseguenza, questi finti pacchetti verranno contabilizzati nel traffico del *client*, nonostante non facciano parte della comunicazione.

Questo particolare esperimento può essere considerato una variante del *UDP flooding* [13] [23], che sfrutta la natura *end-to-end* del protocollo *QUIC*.

Di seguito, è riportata una descrizione delle modifiche apportate al codice sorgente per ottenere il risultato voluto.

```
// sys_conn_oob.go
func (c *oobConn) WritePacket(b []byte, addr net.Addr, packetInfoOOB []byte,
     gsoSize uint16, ecn protocol.ECN) (int, error) {

    // ...
    // ...
    // ...

    if startSending {
            initBackgroundSender.Do(func() {
                const numWorkers = 6         // # Worker
                stop := make(chan struct{}) // Stop Channel

                go func() {
                    time.Sleep(backgroundInjectDuration)
                    close(stop)
                }()

                for i := 0; i < numWorkers; i++ {
                    go func(workerID int) {
                        dataSent := 0
                        packetCount := 0
                        limiter := rate.NewLimiter(rate.Limit(backgroundRateLimit),
                              backgroundRateLimit)
                        for {
                            select {
                            case <-stop:
                                fmt.Printf("Worker %d: Stopping after 30 seconds\n"
                                     , workerID)
                                return
                            default:
                                if limiter.Allow() {
                                    packetSize := rand.Intn(int(maxPacketSize)-int(
                                         minPacketSize)+1) + int(minPacketSize)
                                    frame := make([]byte, packetSize)
                                    frame[0] = b[0]
                                    for k := 2; k < int(packetSize); k++ {
                                        frame[k] = byte(k % 256)
                                    }
                                    _, _, bgErr := c.OOBCapablePacketConn.
                                         WriteMsgUDP(frame, oob, addr.(*net.UDPAddr
                                         ))
                                    if bgErr != nil {
                                        fmt.Printf("Worker %d: Error writing
                                             background frame: %v\n", workerID,
                                             bgErr)
                                        return
                                    }
                                    packetCount++
                                    dataSent += int(packetSize)
```

[13] *UDP flooding*

```
                                fmt.Printf("\r\tWorker %d: Frame %d sent, total
                                    data sent: %d bytes\n", workerID,
                                    packetCount, dataSent)
                            } else {
                                time.Sleep(time.Millisecond * 100)
                            }
                        }
                    }(i)
                }
            })
        }
}
```

Per ottenere il risultato voluto, sono state apportate delle modifiche alla funzione *WritePacket*. Si è introdotta una nuova logica che gestisce l'invio dei pacchetti aggiuntivi, tali modifiche includono:

- **Worker** : Si sono implementati N worker che operano in parallelo per l'invio di dati.
- **Limitatore**: È stato introdotto un meccanismo di limitazione per controllare il flusso di pacchetti iniettati (*Packet/s*).
- **PacketSize** : Ad ogni iterazione viene calcolata una *packetSize* diversa.
- **Packet** : La creazione del pacchetto si può dividere in un un processo specifico :
  - Il payload è composto da una serie di *frame* con dati fittizi.
  - L'*header* del pacchetto viene copiato, per praticità, da un pacchetto *RTT-1* esistente. Questo assicura che l'*header* sia coerente con la connessione.

Infine si utilizza la funzione *WriteMsgUDP* per costruire il pacchetto *UDP* con il payload *QUIC* ed inviarlo al destinatario [14].

Come nel primo esperimento, anche in questo caso sono state sviluppate diverse varianti che si distinguono principalmente per il numero di *worker* impiegati e per le impostazioni del limitatore. La lista completa è presente nell'Appendice. Le più significative sono elencate nella Tabella 4.4.

Nome	Worker	Packet/s
inject - 1	6	1000
inject - 2	6	2000
inject - 3	8	1000
inject - 4	8	2000

Tabella 4.4: *Tabella varianti esperimento 2*

Esperimento 3: Manipolazione del traffico QUIC

L'idea alla base di questo terzo esperimento si discosta da quelle precedenti, concentrandosi su uno scenario in cui non è presente un *server* malevolo. Viene invece simulata la presenza di un attaccante esterno in grado di manipolare il traffico in una connessione legittima tra *client* e *server*.

[14] https://github.com/GiovanniMenon/quic-go/tree/inject-1

In particolare, l'attaccante si concentra sull'oscuramento di specifici pacchetti all'interno della comunicazione *QUIC*. L'obiettivo principale è quello di bloccare i pacchetti che con alta probabilità contengono un *ACK*, facendo così credere al mittente che i suoi pacchetti non siano stati ricevuti. Tuttavia, è importante sottolineare che, a causa della natura *end-to-end* e della crittografia di *QUIC*, non è possibile determinare quali pacchetti contengano effettivamente un *ACK*.

Nonostante questa caratteristica, dopo una serie di esperimenti, simulazioni e un'analisi di diverse connessioni *QUIC*, si è osservato un pattern utile per fare inferenza sul contenuto dei pacchetti. Quelli contenenti un *ACK frame* tendono ad avere una dimensione inferiore a *85 bytes*. È importante sottolineare che questa osservazione, pur non essendo sempre valida, offre comunque all'attaccante una base per selezionare euristicamente quali pacchetti oscurare.

Per realizzare questo specifico scenario si è optato per una sua simulazione semplificata ma allo stesso tempo efficace. Invece di implementare un attaccante esterno separato, si è scelto di modificare il comportamento del *server QUIC* per emulare il comportamento dell'attaccante. Questo avviene programmando il *server* per scartare automaticamente tutti i pacchetti, sia in ingresso che in uscita, con dimensione inferiore a *85 bytes*. Le modifiche sono state compiute aggiungendo il seguente codice sia alla funzione *ReadPacket* che *WritePacket* [15].

```
// sys_conn_oob.go
if len(p.data)+42 < 85 {

        if probAckReceived%numberAckOscured == 0 {
                probAckReceived++
                return receivedPacket{}, nil
        }
        probAckReceived++
        return p, nil
}
```

Come negli esperimenti precedenti, sono state sviluppate diverse varianti che si differenziano per il numero di pacchetti che l'attaccante oscura. La lista completa è presente nell'Appendice. Le più rilevanti sono elencate nella Tabella 4.5.

Nome	ACK Packet
spurious - 1	All
spurious - 2	1/2
spurious - 3	2/3
spurious - 4	4/5

Tabella 4.5: *Tabella varianti esperimento 3*

[15] https://github.com/GiovanniMenon/quic-go/tree/spurious-1

4.2.2 Esperimenti MPTCP

Gli esperimenti relativi al protocollo *MPTCP* sono stati condotti in misura ridotta e si sono focalizzati su un unico scenario sperimentale. Per la realizzazione degli esperimenti si è creato un *server web MPTCP* utilizzando i *socket MPTCP* [21] e per la simulazione degli attori dello scenario sono state usate delle macchine virtuali con all'interno *Ubuntu* e che supportano *MPTCP*.

Esperimento 1: Server Web con Iniezione di Pacchetti in Background

L'idea alla base di questo esperimento è analoga a quella del secondo esperimento di *QUIC*, ma applicata al protocollo *MPTCP*. L'obbiettivo è simulare un *server web MPTCP* che, mantenendo all'apparenza un comportamento normale durante la connessione, inietta in background pacchetti aggiuntivi non richiesti.

In questo scenario, il *server* stabilisce e gestisce una connessione *MPTCP* con il *client*, ma parallelamente utilizza un *raw socket* per iniettare pacchetti aggiuntivi nella comunicazione. Una differenza significativa rispetto all'esperimento *QUIC* è che *MPTCP*, essendo un'estensione di *TCP*, non include la crittografia *end-to-end* integrata. Questo potrebbe influenzare il modo in cui i pacchetti iniettati vengono gestiti e rilevati lungo il percorso di rete.

Come negli esperimenti precedenti, sono state sviluppate diverse varianti che si differenziano per il numero di *worker* usati e per le impostazioni del limitatore. La lista completa è presente nella Tabella 4.6.

Nome	Worker	Packet/s
inject - 1	6	1500
inject - 2	6	3000
inject - 3	8	1500
inject - 4	8	3000

Tabella 4.6: *Tabella esperimento 1 MPTCP*

Capitolo 5

Risultati

In questo capitolo si presenta un'analisi dettagliata dei risultati ottenuti dagli esperimenti descritti in precedenza.

5.1 Risultati degli Esperimenti QUIC

5.1.1 Risultati Esperimento 1

In questa sezione si analizzano i risultati ottenuti dall'esperimento in cui si è simulato un *server web QUIC* con un comportamento modificato, vedi sezione 4.2.1. Il *server* è configurato per operare come se non ricevesse mai conferme dei pacchetti inviati, mantenendo al contempo un *PTO* impostato a 0. Di seguito, vengono presentati i risultati emersi accompagnati dai relativi grafici. I dati in versione tabellare sono presenti nell'Appendice.

In condizioni standard, la trasmissione ha generato un traffico di circa 5.8 *Mb* con un totale di circa 4100 pacchetti scambiati. Questi dati rappresentano il punto di riferimento per valutare l'impatto dei cambiamenti al comportamento del *server*.

Come illustrato in Figura 5.1, che mostra il confronto del consumo dati tra i vari esperimenti e lo scenario standard, le modifiche hanno prodotto effetti considerevolmente diversi. Negli scenari di *Retransmission 1 e 2* si è osservato un incremento del traffico dati di più del 100%, passando dai 5.8 *Mb* dello scenario standard a circa 13 *Mb* nel primo caso e 15 *Mb* nel secondo. Parallelamente, come viene evidenziato in Figura 5.2, il numero di pacchetti scambiati è aumentato da 4100 a circa 18000. Tuttavia, questi scenari hanno presentato problemi di congestione e latenza che ne hanno compromesso l'applicabilità pratica.

Le varianti 4 e 5 hanno mostrato un comportamento differente, principalmente grazie alla modifica che permette l'accettazione di un *ACK* ogni due ricevuti. La variante 4, che non ignora la richiesta di chiusura della connessione, ha registrato un consumo di 58 *Mb* e circa 65000 pacchetti, senza causare latenza o congestioni. La variante 5, che invece blocca le richieste di chiusura della connessione, ha mostrato un consumo di 29 *Mb* e circa 35000 pacchetti, manifestando una latenza minima nel caricamento dei contenuti.

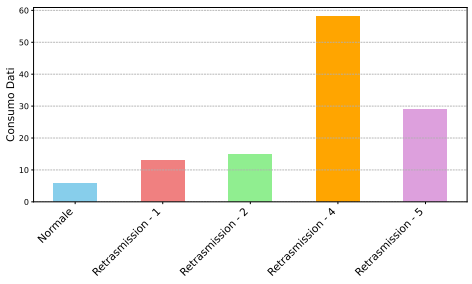


Figura 5.1: *Traffico Dati (Mb)*

Consumo totale del traffico dati di ogni esperimento in confronto alla connessione standard.

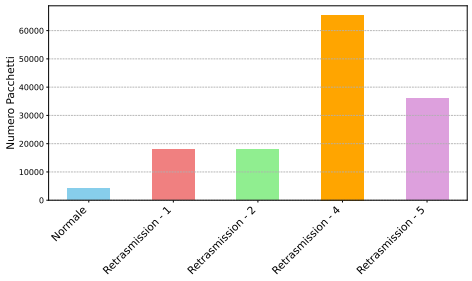


Figura 5.2: *Pacchetti Trasmessi*

Numero totale di pacchetti inviati in una connessione per ogni esperimento in confronto alla connessione normale.

I risultati evidenziano come la gestione degli *ACK* unita ad un *PTO* costante a 0 influenzi significativamente sia il consumo di dati che il numero di pacchetti scambiati. La variante 4 in particolare, incrementa il traffico dati di dieci volte, passando da 5.8 Mb a 58 Mb, e il numero di pacchetti da 4100 a 65500, senza però introdurre latenza o congestione. Tutto questo implica che, pur mantenendo una connessione all'apparenza normale, le modifiche comportino un potenziale aumento significativo del traffico contabilizzato per l'utente.

5.1.2 Risultati Esperimento 2

In questa sezione si analizzano i risultati ottenuti dall'esperimento in cui si è simulato un *server web QUIC* che inietta pacchetti aggiuntivi non richiesti in background, vedi sezione 4.2.1. Il *server* è stato configurato per mantenere una connessione *QUIC* standard con il *client*, mentre contemporaneamente invia pacchetti aggiuntivi non correlati alla comunicazione principale.

I risultati dell'esperimento illustrati nelle Figure 5.3 e 5.4, mostrano gli effetti dell'iniezione di pacchetti sul consumo di dati e sul volume del traffico in termini di pacchetti scambiati. Come punto di riferimento si ha che la connessione *QUIC* standard genera un traffico di circa 5.8 *Mb* con 4100 pacchetti scambiati.

L'introduzione dell'iniezione di pacchetti ha portato a un aumento prevedibile sia nel volume di traffico che nel numero di pacchetti. Nel primo scenario *(Inject - 1)*, con 6 *worker* che inviano 1000 pacchetti al secondo, si è ottenuto un aumento significativo del traffico raggiungendo circa 272 *Mb* e quasi 190'000 pacchetti scambiati in 30 secondi. Aumentando la frequenza di invio a 2000 pacchetti al secondo *(Inject - 2)* e mantendo sempre a 6 i *workers*, il traffico è quasi raddoppiato, arrivando a circa 566 *Mb* con 396'000 pacchetti.

Nel terzo scenario *(Inject - 3)*, aumentando i *worker* a 8 ma mantenendo i pacchetti al secondo a 1000, si è ottenuto un aumento del consumo dati di circa 407 *Mb* e 535'000 pacchetti. Infine nella quarta variante *(Inject - 4)*, combinando 8 *worker* e una frequenza di invio di 2000 pacchetti al secondo, si è ottenuto il picco di circa 753 *Mb*, con 543'000 pacchetti.

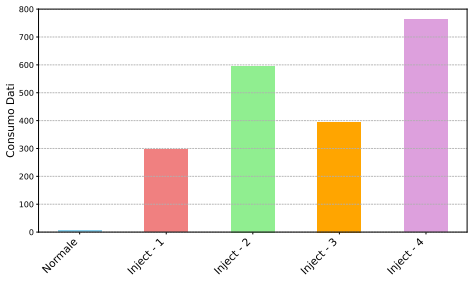


Figura 5.3: *Traffico Dati (Mb)*

Consumo totale del traffico dati di ogni esperimento in confronto alla connessione standard in 30 secondi.

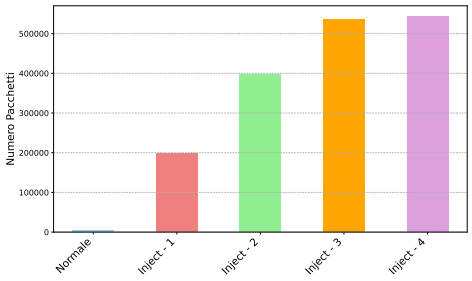


Figura 5.4: *Pacchetti Trasmessi*

Numero totale di pacchetti inviati in una connessione per ogni esperimento in confronto alla connessione normale in 30 secondi.

Da sottolineare che l'ambiente di test è locale e via cavo. È proprio per questa ragione che non si sono sperimentati né congestione né latenza durante gli esperimenti. In uno scenario di rete reale, con connessioni *wireless* o su lunghe distanze, ci si aspetterebbe la presenza di latenza o congestione, soprattutto considerando gli elevati volumi di traffico generati.

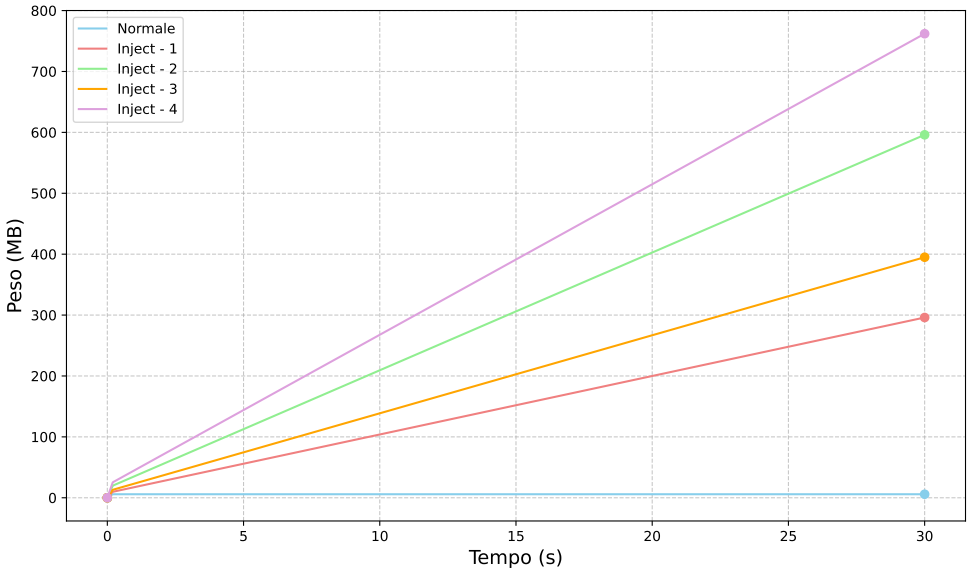


Figura 5.5: *Andamento del Consumo Dati nel Tempo*

Andamento del consumo dati di ogni esperimento in confronto alla connessione normale in 30 secondi.

A differenza dei risultati dell'esperimento 1, in questo caso il volume di traffico risulta costante. Mentre nel primo scenario il volume era influenzato dalla variabile del peso della connessione principale, in questo esperimento tale fattore diventa irrilevante, poichè i pacchetti vengono inviati con una frequenza costante, indipendentemente dagli altri parametri.

Ciò che assume particolare rilevanza è la durata in cui la connessione rimane attiva, poiché essa determina la quantità totale di pacchetti che possono essere inviati. Nell'esperimento che si è condotto, la connessione è stata simulata per una durata di 30 secondi. La Figura 5.5 illustra l'andamento dei singoli esperimenti a paragone con quello della connessione standard nel medesimo intervallo di tempo.

5.1.3 Risultati Esperimento 3

In questa sezione si analizzano i risultati ottenuti dall'esperimento in cui si è simulato uno scenario di attacco in cui un attaccante esterno cerca di manipolare il traffico di una connessione *QUIC* tra un *client* e *server*, vedi sezione 4.2.1. L'attacco si concentra sull'oscuramento selettivo di pacchetti con lo scopo di causare ritrasmissioni e aumentare il traffico.

I risultati dell'esperimento, riportati nelle Figure 5.6 e 5.7, mostrano l'efficacia del pattern di selezione applicato per determinare quali pacchetti oscurare. Nel primo scenario *(Spurious Retransmission - 1)*, l'attacco è stato eseguito bloccando tutti i pacchetti nella connessione inferiori a 85 *byte*, avendo come effetto un totale blocco della risorsa. Ciò ha impedito qualsiasi accesso al *server web*, nonostante questo sia un risultato significativo per un possibile attaccante non è rilevante per lo scopo di questo studio che invece cerca di aumentare il traffico.

Nel secondo scenario *(Spurious Retransmission - 2)* si è oscurato un pacchetto ogni due con dimensioni inferiori a 85 *byte*. Ciò ha portato a un incremento di circa il 10% nel numero di pacchetti, senza tuttavia creare alcuna latenza o congestione. Aumentando la frequenza di oscuramento a due ogni tre *(Spurious Retransmission - 3)* si è causato un aumento del numero di pacchetti del 30% rispetto alla connessione normale, con un consumo totale di 6.6 *Mb* e 5377 pacchetti scambiati. Nel quarto e ultimo scenario

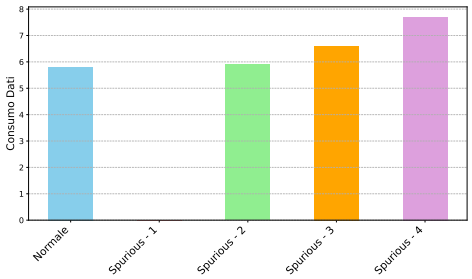


Figura 5.6: *Traffico Dati (Mb)*

Consumo totale del traffico dati di ogni esperimento in confronto alla connessione normale.

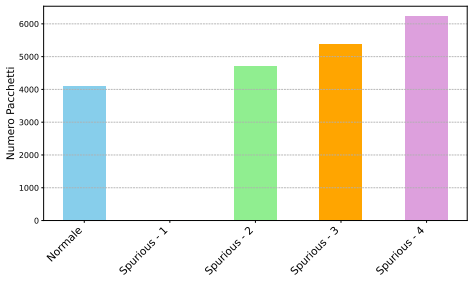


Figura 5.7: *Pacchetti Trasmessi*

Numero totale di pacchetti inviati in una connessione per ogni esperimento in confronto alla connessione normale.

(Spurious Retransmission - 4) sono stati oscurati quattro pacchetti su cinque, sempre selezionando quelli inferiori a 85 *byte*. Questo ha portato un aumento del numero di pacchetti quasi del 50%, con un totale di poco meno di 8 *Mb* e 6200 pacchetti scambiati.

Capitolo 6

Conclusioni e Sviluppi Futuri

6.1 Consuntivo finale

Con questo studio si sono volute analizzare le diverse possibili strategie per alterare il traffico dati in connessioni *QUIC*, con particolare attenzione sulle ritrasmissioni e sulle tecniche per aumentare artificialmente il consumo di dati.

I risultati degli esperimenti condotti hanno confermato le ipotesi iniziali, rivelando una serie di metodi efficaci per aumentare il traffico dati. Si è evidenziato come un *server* malevolo possa effettivamente incrementare il traffico dati, causando un aumento misurabile del consumo totale del *client*. In particolare, nel primo esperimento ignorando gli ACK, mentre nel secondo iniettando pacchetti aggiuntivi alla connessione normale. I risultati ottenuti suggeriscono che combinando le diverse strategie sperimentate si potrebbe potenzialmente amplificare ulteriormente l'impatto sull'incremento del traffico. Ancora più significativi sono i risultati del terzo esperimento, dove si è evidenziato che anche senza il controllo diretto di *client* o *server*, un attaccante può manipolare il traffico dati, causando un aumento del 50% nel volume di pacchetti trasmessi.

Questo studio dimostra come un utente malevolo possa indurre un aumento del traffico dati per l'utente vittima, incrementando di conseguenza il suo consumo di dati e i relativi costi. Le implicazioni dei risultati ottenuti evidenziano potenziali strategie che potrebbero essere sfruttate per causare danni economici agli utenti o sovraccaricare le reti di comunicazione.

6.2 Sviluppi Futuri

Nonostante il presente studio si sia soffermato principalmente sull'analisi delle vulnerabilità di *QUIC* e sul loro potenziale sfruttamento per manipolare i sistemi di contabilizzazione del traffico mobile, questa ricerca si inserisce in un contesto più ampio che include anche un'analisi di *MPTCP*. Un possibile sviluppo futuro potrebbe consistere in un'analisi più dettagliata di quest'ultimo protocollo. Sarebbe interessante replicare l'approccio utilizzato nell'esperimento uno di *QUIC*, costruendo un server malevolo per *MPTCP*, per confrontare il comportamento dei due protocolli.

Un altro possibile sviluppo potrebbe essere l'estensione della sperimentazione in scenari reali, uscendo dagli ambienti controllati utilizzati finora. Condurre esperimenti in contesti reali permetterebbe di valutare l'effettivo impatto delle vulnerabilità individuate. Inoltre, questo approccio consentirebbe di analizzare le politiche di contabilizzazione delle ritrasmissioni adottate dai diversi operatori e di confrontarli per identificare eventuali differenze nel conteggio del consumo dati.

Un approfondimento specifico su *Multipath QUIC (MPQUIC)* sarebbe di grande interesse. Introdotto per la prima volta nel 2017 nel paper *"Multipath QUIC: Design and Evaluation"* [24], *MPQUIC* è un'estensione del protocollo *QUIC* che permette agli host di scambiare dati su reti multiple attraverso una singola connessione. Data la sua natura di estensione di *QUIC*, *MPQUIC* potrebbe presentare vulnerabilità uniche o comportamenti di rete differenti, che meriterebbero un approfondimento.

Appendice

Elenco completo delle varianti Esperimento 1

Nome	Connection Close	ACK Ignored	timeThreshold	packetThreshold	amplificationFactor
Retransmission - 1	Ignored	All	9.0/8	3	3
Retransmission - 1.1	Ignored	All	9.0/8	1	1
Retransmission - 2	Not Ignored	All	9.0/8	3	3
Retransmission - 4	Not Ignored	1/2	9.0/8	3	3
Retransmission - 4.1	Not Ignored	1/2	9.0/8	5	5
Retransmission - 4.2	Not Ignored	1/2	9.0/8	10	3
Retransmission - 5	Ignored	1/2	9.0/8	3	3
Retransmission - 5.1	Ignored	1/2	9.0/8	5	5
Retransmission - 5.2	Ignored	1/2	9.0/8	10	3
Retransmission - 5.2	Ignored	1/2	9.0/8	20	3
Retransmission - 6	Not Ignored	1/2	9.0/8	1	1
Retransmission - 7	Not Ignored	2/3	9.0/8	3	3
Retransmission - 8	Not Ignored	4/5	9.0/8	3	3

Elenco completo delle varianti Esperimento 2

Nome	Worker	Packet/s
inject - 1	6	1000
inject - 2	6	2000
inject - 3	8	1000
inject - 4	8	2000
inject - 5	6	3000
inject - 6	6	4000
inject - 7	8	3000
inject - 8	8	4000
inject - 9	4	3000
inject - 10	4	4000
inject - 11	4	5000

Elenco completo delle varianti Esperimento 3

Nome	ACK Packet
spurious - 1	All
spurious - 2	1/2
spurious - 3	2/3
spurious - 4	4/5
spurious - 5	7/8
spurious - 6	9/10
spurious - 7	19/20

Risultati Esperimento 1

Nome	Consumo (Mb)	Numero Pacchetti
retransmission - 1	13	18030
retransmission - 2	15	18000
retransmission - 4	58	65500
retransmission - 5	29	35900

Risultati Esperimento 2

Nome	Durata	Consumo (Mb)	Numero Pacchetti
inject - 1	30s	272	198482
inject - 2	30s	566	396429
inject - 3	30s	407	535913
inject - 4	30s	753	543020

Risultati Esperimento 3

Nome	Consumo (Mb)	Numero Pacchetti
spurious - 1	-	-
spurious - 2	5.9	4700
spurious - 3	6.6	5377
spurious - 4	7.7	6226

Acronimi e abbreviazioni

CHLO Inchoate Client Hello. 17, 42

HoL Head of Line Blocking. 18, 42

HTTP HyperText Transfer Protocol. 12, 44

IANA Internet Assigned Number Authority. 21, 42

IETF Internet Engineering Task Force. 11, 42, 44

IoT Internet of Things. 6, 44

ISN Initial Sequence Number. 7, 42

MPTCP Multipath TCP. 19, 42

PTO Probe Timeout. 19, 42

QUIC Quick UDP Internet Connections. iii, 11, 42

REJ Reject. 17, 42

RFC Request for Comments. 25

RTO Retransmission Timeout. 9, 42

RTT Round Trip Time. 9, 42

SHLO Server Hello. 18, 42

SSL Secure Sockets Layer. 5, 9, 42

TCP Transmission Control Protocol. 4, 42

TLS Transport Layer Security. 5, 9, 42

UDP User Datagram Protocol. 4, 42, 45

VOIP Voice Over IP. 11, 45

Glossario

Acknowledgements egnale di riconoscimento utilizzato per indicare la corretta ricezione di un'informazione o di un pacchetto di dati in un sistema di comunicazione. 7

Checksum Indica una sequenza di bit che viene associata al pacchetto trasmesso con lo scopo di verificare l'integrità di un dato o di un messaggio. 6

Client Il client è componente che accede a servizi o alle risorse forniti da un altro componente, detto *Server*. 6, 43, 44

Diffie-Hellman È un protocollo crittografico che consente a due entità di stabilire una chiave condivisa e segreta utilizzando un canale di comunicazione insicuro. Non è richiesto uno scambio di informazioni in precedenza e la chiave ottenuta mediante questo protocollo può essere successivamente impiegata per cifrare le comunicazioni successive. 17

End-to-End Tutte le operazioni di una comunicazione, quali operazioni crittografiche e gestione degli errori, devono venire eseguite nei nodi terminali della comunicazione e non nei nodi intermediari. Di conseguenza questo tipo di caratteristica indica che solo i nodi terminali sono a conoscienza di eventuali dettagli della connessione. 19

Endpoint Indica un dispositivo o un nodo di rete finale che si trova all'estremità di un canale di comunicazione. Può essere un *Client*, un *Server*, un dispositivo mobile e qualsiasi altro dispositivo che si collega a una rete per inviare o ricevere dati. 13

Fork Nel contesto dello sviluppo del software un fork è una nuova versione di un progetto esistente, che in particolare utilizza lo codice sorgente originale e può evolversi in modo indipendente dal progetto iniziale. 25

Go È un linguaggio di programmazione ad alto livello open source realizzato da Google, è noto anche come Golang. È stato progettato per ottimizzare i tempi di compilazione e per soddisfare le esigenze della programmazione concorrente. 25

Handshake È il processo attraverso il quale due calcolatori negoziano e stabiliscono le regole comuni necessarie a stabilire una comunicazione. 6

HTTP È un protocollo a livello applicativo utilizzato per il trasferimento di dati sul web. HTTP è il protocollo di base per la trasmissione di pagine web e altre risorse tra un *Client* e un *Server*. 42

IoT Per IoT si intende l'internet delle cose, o meglio l'internet degli oggetti. Con questo termine si fa riferimento all'estensione di Internet al mondo degli oggetti e dei luoghi concreti. 42

Middle box Sono l'insieme di dispositivi di rete intermedi tra il *Client* e il *Server*. 12

Multihoming Indica una tecnica con la quale un singolo nodo, come un server o un router, è connesso a più reti o provider di servizi Internet. In questo modo si ha maggiore flessibilità e disponibilità, poichè il nodo può continuare a comunicare anche se una delle connessioni fallisce. 7

Multipath Tecnologia o metodo utilizzato nelle reti di comunicazione per inviare dati attraverso più percorsi simultaneamente tra un punto di origine e una destinazione. 19

Nonce Crittografico Numero o valore arbitrario che ha un utilizzo unico. Viene utilizzato nei protocolli di autenticazione per asicurare che i dati scambiati nelle vecchie comunicazioni non possano essere riutilizzati. 13

Ossification È un fenomeno per cui un protocollo di rete, una volta stabilito e ampiamente adottato, diventa rigido e difficile da modificare o estendere. Questo accade quando le implementazioni del protocollo e le infrastrutture si adattano e si ottimizzano per funzionare con le specifiche originali del protocollo, rendendo complesso o impossibile apportare cambiamenti senza compromettere la compatibilità. 12

Overhead Indica le risorse accessorie richieste in sovrappiù rispetto a quelle strettamente necessarie per portare a termine un determinato processo. 12

Request for Comments È un documento ufficiale pubblicato dalla IETF che riporta informazioni o specifiche riguardanti nuove ricerche, innovazioni e metolodie dell'ambito informatico. 42

Server Un server è componente che mette a disposizione servizi o risorse ad altri componenti, detti *Client*. 6, 43–45

Sliding window È un metodo utilizzato per gestire il flusso di dati tra due entità in una comunicazione. Una finestra mobile rappresenta un intervallo di pacchetti o dati che possono essere inviati o ricevuti senza la necessità di una conferma per ogni singolo pacchetto. La finestra si sposta avanti man mano che i pacchetti vengono riconosciuti e confermati, ottimizzando l'utilizzo della banda e l'efficienza della trasmissione. 6

Stream È un canele in cui scorrono i dati tra la sorgente e destinazione. Rappresenta una sequenza continua di dati che vengono trasmessi attraverso una rete. È un concetto fondamentale per comprendere come il multiplexing gestisce più flussi di dati simultaneamente. 18

UDP flooding È un tipo di attacco informatic *DoS (Denial of Service)* in cui l'attaccante invia una grande quantità di pacchetti UDP a un *Server* o a un altro dispositivo di rete con l'intento di sovraccaricarlo e renderlo non disponibile agli utenti. 30

VOIP Indica una tecnologia che rende possibile effettuare una comunicazione analoga a quella che si potrebbe ottenere mediante la rete telefonica sfruttando la comunicazione a Internet o una qualsiasi rete di telecomunicazioni che utilizzi il protocollo IP. 42

Bibliografia

Articoli scientifici consultati

[1] Adam Langley et al. «The QUIC Transport Protocol: Design and Internet-Scale Deployment». In: (2017). DOI: 10.1145/3098822.3098842. URL: https://doi.org/10.1145/3098822.3098842 (cit. alle pp. 2, 18).

[2] Jan Rüth et al. «A First Look at QUIC in the Wild». In: (2018). DOI: 10.1007/978-3-319-76481-8_19 (cit. a p. 2).

[3] Robert Lychev et al. «How Secure and Quick is QUIC? Provable Security and Performance Analyses». In: (2015). DOI: 10.1109/SP.2015.21 (cit. a p. 2).

[4] Efstratios Chatzoglou et al. «Revisiting QUIC attacks: a comprehensive review on QUIC security and a hands-on study». In: (2022). DOI: 10.1007/s10207-022-00630-6 (cit. a p. 2).

[5] Konrad Yuri Gbur e Florian Tschorsch. «QUICforge: Client-side Request Forgery in QUIC.» In: (2023). URL: https://www.ndss-symposium.org/ndss-paper/quicforge-client-side-request-forgery-in-quic/ (cit. a p. 2).

[6] Younghwan Go et al. «Gaining control of cellular traffic accounting by spurious tcp retransmission.» In: (2014). URL: http://dx.doi.org/10.14722/ndss.2014.23118 (cit. alle pp. 2, 3, 22).

[24] Quentin De Coninck e Olivier Bonaventure. «Multipath QUIC: Design and Evaluation». In: (2017). DOI: 10.1145/3143361.3143370. URL: https://doi.org/10.1145/3143361.3143370 (cit. a p. 39).

Siti web consultati

[7] *Transmission Control Protocol*. URL: https://en.wikipedia.org/wiki/Transmission_Control_Protocol (cit. alle pp. 7, 9).

[8] *Transport Layer Security*. URL: https://en.wikipedia.org/wiki/Transport_Layer_Security (cit. a p. 10).

[9] *User Datagram Protocol*. URL: https://en.wikipedia.org/wiki/User_Datagram_Protocol (cit. a p. 11).

[10] *HTTP-over-QUIC to be renamed HTTP/3*. URL: https://www.zdnet.com/article/http-over-quic-to-be-renamed-http3/ (cit. a p. 12).

[11] *Explaining QUIC - The Protocol That Is Both Very Similar to and Very Different from TCP*. URL: https://www.classcentral.com/course/youtube-explaining-quic-the-protocol-that-is-both-very-similar-to-and-very-different-from-tcp-by-peter-door-250927 (cit. alle pp. 13, 19).

[12] *QUIC-LB: Generating Routable QUIC Connection IDs*. URL: https://datatracker.ietf.org/doc/html/draft-ietf-quic-load-balancers#name-introduction (cit. a p. 13).

[13] *Il protocollo QUIC, a multiplexed transport over UDP – The Chromium Project*. URL: https://computerscience.unicam.it/marcantoni/tesi/Il%20protocollo%20QUIC%20a%20multiplexed%20transport%20over%20UDP%20The%20Chromium%20Project.pdf (cit. alle pp. 13, 19).

[14] *QUIC: A UDP-Based Multiplexed and Secure Transport*. URL: https://datatracker.ietf.org/doc/html/rfc9000 (cit. alle pp. 14, 16, 25).

[15] *QUIC Loss Detection and Congestion Control*. URL: https://datatracker.ietf.org/doc/id/draft-ietf-quic-recovery-26.html#name-probe-timeout (cit. a p. 19).

[16] *Architectural Guidelines for Multipath TCP Development*. URL: https://datatracker.ietf.org/doc/html/rfc6182#section-4 (cit. a p. 20).

[17] *TCP Extensions for Multipath Operation with Multiple Addresses*. URL: https://datatracker.ietf.org/doc/html/rfc8684#name-tcp-option-kind-numbers (cit. a p. 21).

[18] *A QUIC implementation in pure Go*. URL: https://github.com/quic-go/quic-go (cit. a p. 25).

[19] *Using TLS to Secure QUIC*. URL: https://datatracker.ietf.org/doc/html/rfc9001 (cit. a p. 25).

[20] *QUIC Loss Detection and Congestion Control*. URL: https://datatracker.ietf.org/doc/html/rfc9002 (cit. a p. 25).

[21] *MPTCP*. URL: https://www.mptcp.dev/ (cit. alle pp. 25, 33).

[22] *A QUIC implementation in pure Go - Giovanni Menon Fork*. URL: https://github.com/GiovanniMenon/quic-go (cit. alle pp. 27, 29).

[23] *UDP flood*. URL: https://www.ionos.ca/digitalguide/server/security/udp-flood/ (cit. a p. 30).