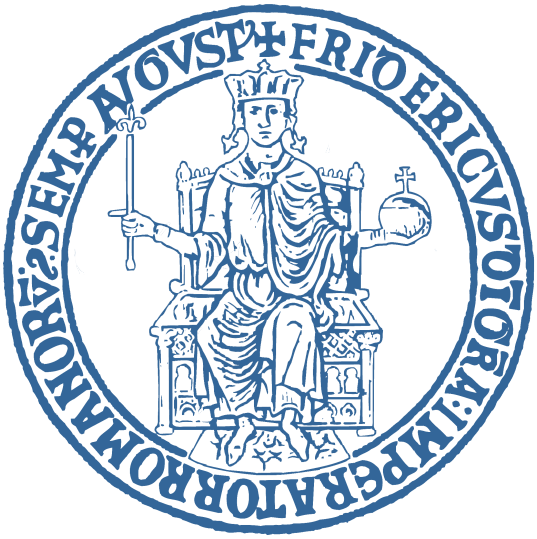

Università degli Studi di Napoli Federico II

Async Traceroute Library/Command-line Tool in Rust

2023/2024

Professore	*Studente*
Simon Pietro Romano	Vincenzo Tramo

Indice

1 Introduzione

La seguente relazione ha l'obiettivo di documentare una implementazione del noto programma `traceroute` scritta con il linguaggio di programmazione Rust. Gli obiettivi principali del progetto sono:

- Implementare le principali funzionalità del classico `traceroute`, in particolare:
  - Specificare il numero massimo di hops (max TTL - Time To Live);
  - Definire il numero di probes da inviare per ogni hop;
  - Impostare un timeout specifico per ogni probe;
  - Supportare diversi probe methods (in particolare: UDP, TCP, ICMP);
  - Specificare la porta di destinazione (per UDP o TCP);
  - Specificare il sequence number iniziale (per ICMP);
  - Abilitare o disabilitare reverse DNS lookups;
  - Specificare l'interfaccia di rete da utilizzare.
- Progettare il sistema in modo da facilitare l'implementazione e l'integrazione di nuovi probe methods;
- Fornire una libreria asincrona di facile utilizzo, integrabile in altri progetti;
- Fornire un'utility utilizzabile da linea di comando.

1.1 Traceroute

Traceroute è uno strumento di diagnostica di rete utilizzato per tracciare il percorso che i pacchetti IP percorrono da un host sorgente a una destinazione specifica. Originariamente sviluppato da Van Jacobson nel 1987, traceroute è diventato uno degli strumenti di rete più utilizzati per identificare problemi di rete e analizzare la topologia della rete.

1.1.1 Come funziona Traceroute

Il funzionamento di traceroute si basa sull'utilizzo del protocollo ICMP (Internet Control Message Protocol) e dei meccanismi di `TTL` (Time To Live). Il `TTL` è un campo nell'header IP che specifica il numero massimo di salti (hop) che un pacchetto può attraversare prima di essere scartato. In particolare, Traceroute utilizza il campo `TTL` per sollecitare un messaggio **ICMP Time Exceeded** da ciascun router lungo il percorso. Ogni router che gestisce il pacchetto diminuisce il valore del campo `TTL`, che agisce di fatto come un "hop counter". Quando un router riceve un IP datagram con il campo `TTL` impostato a 0, risponde con un messaggio `ICMP Time Exceeded` che rivela il suo indirizzo IP.

1.1.2 Traceroute Probe Methods

Esistono diversi traceroute probe methods, ognuno dei quali è stato progettato per funzionare meglio in uno scenario in cui un altro fallisce. La Figura 1 mostra come funziona il classico traceroute basato su UDP.

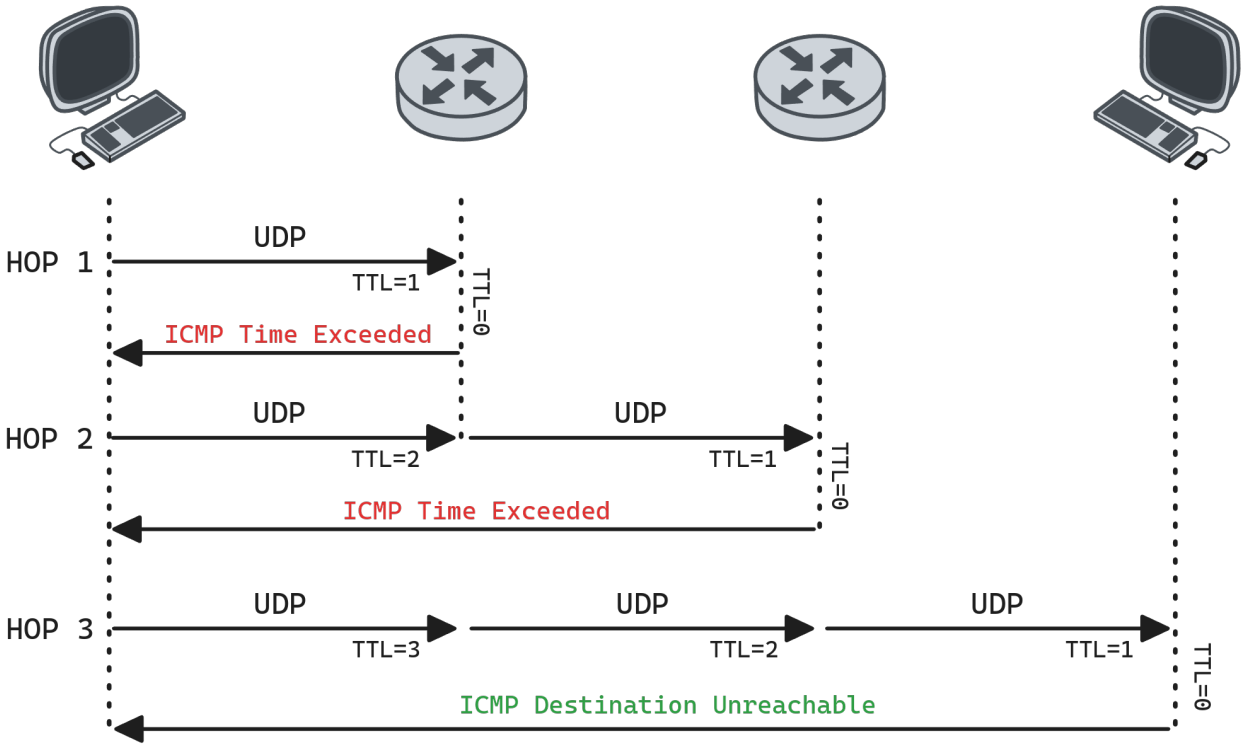

Figura 1: Illustrazione funzionamento del classico traceroute basato su UDP

Traceroute invia una serie di UDP datagram con le seguenti caratteristiche:

- una porta di destinazione improbabile (ossia, una porta su cui è improbabile che l'host di destinazione sia in ascolto) che viene incrementata di 1 per ogni nuovo UDP datagram inviato
- con un valore `TTL` incrementale (partendo da 1)

Il valore `TTL` determina il numero massimo di hop (router) che un pacchetto può attraversare prima di essere scartato da un router. Quando il `TTL` di un pacchetto arriva a 0, il router che lo riceve scarta il pacchetto e risponde con un messaggio `ICMP Time Exceeded`, rivelando il proprio indirizzo IP. Se, invece, il datagramma UDP raggiunge l'host di destinazione, quest'ultimo risponderà con un messaggio `ICMP Destination Unreachable`, rivelando il proprio indirizzo IP e segnalando al processo `traceroute` sull'host sorgente che la destinazione è stata raggiunta.

1.2 Perché Rust

La caratteristica distintiva di Rust come linguaggio di programmazione è la sua capacità di prevenire l'accesso a dati non validi a tempo di compilazione. Studi di ricerca hanno evidenziato che i problemi relativi all'accesso a dati non validi rappresentano circa il 70% dei gravi bug di sicurezza[1]. Rust elimina questa classe di bug, garantendo che il programma sia memory-safe senza imporre alcun costo a runtime. In Rust, è impossibile accedere a runtime a riferimenti che sono diventati invalidi durante il corso del programma (a meno che non si faccia uso dei cosidetti `unsafe` blocks). Inoltre, a differenza di altri linguaggi di alto livello come Java, Rust riesce a prevenire i buffer overflow senza eseguire nessun codice addizionale a runtime.

1.3 Contributo del presente lavoro

Il seguente lavoro contribuisce alla comunità di Rust introducendo una libreria (nel gergo di Rust, un crate) traceroute asincrona, che attualmente non esiste. La libreria è progettata per essere facilmente integrabile in altri progetti, permettendo, ad esempio, di eseguire più traceroute verso diverse destinazioni in parallelo.

[1] Vedi gli articoli "We need a safer systems programming language" e "Memory safety"

Implementazione

Uno degli obiettivi principali del progetto è astrarre le varie operazioni necessarie per eseguire traceroute, con l'intento di fornire sufficiente flessibilità per introdurre facilmente nuovi probe methods.

Le operazioni astratte e i metodi generici, che variano in base al tipo di probe method utilizzato, includono:

- **Invio di una probe:** invia una probe a un dato indirizzo di destinazione con un determinato `TTL` (Time To Live), utilizzando un protocollo specifico e rispettando un timeout definito (tempo massimo di attesa per una risposta). I parametri aggiuntivi variano a seconda del tipo di probe method: ad esempio, il protocollo UDP richiede una porta di destinazione, mentre questa non è necessaria per il protocollo ICMP (echo requests);
- **Matching delle risposte con le probe requests:** fornisce una funzione che, data una risposta (non necessariamente una risposta ICMP), associa la risposta alla richiesta inviata, permettendo di marcare la richiesta come completata. Questo processo tipicamente implica l'estrazione di informazioni dalla risposta che corrispondono ai dettagli della richiesta originale.

Inoltre, è necessario definire:

- **Intercettazione dei pacchetti ICMP:** un meccanismo per intercettare tutti i pacchetti ICMP in entrata. Questo è fondamentale per il corretto funzionamento di traceroute, poiché i messaggi ICMP di errore generati dai router lungo il percorso sono quelli che consentono di ottenere l'indirizzo IP del router e di identificarlo.

2 Implementazione

In Rust, un package è un insieme di uno o più crates che forniscono un insieme di funzionalità. Un crate è la più piccola quantità di codice che il compilatore Rust considera alla volta. I crates possono contenere moduli e i moduli possono essere definiti in altri file che vengono compilati con il crate. I crates si dividono in binary crates che vengono compilati in eseguibili (contengono una main function) e in library crate (`lib.rs`) che non contengono una main function e non vengono compilati in eseguibili, ma definiscono funzionalità destinate ad essere condivise con più progetti.

Il progetto è un singolo package chiamato `async-traceroute` e ha la seguente struttura:

```
*[main][~/projects/traceroute-rust/src]$ tree
.
|-- lib.rs
|-- main.rs
|-- traceroute
|   |-- async_socket.rs
|   |-- builder.rs
|   |-- probe
|   |   |-- generator.rs
|   |   |-- parser.rs
|   |   |-- sniffer.rs
|   |   |-- task.rs
|   |-- probe.rs
|   |-- terminal.rs
|   |-- utils
|   |   |-- bytes.rs
|   |   |-- dns.rs
|   |   |-- packet_utils.rs
|   |-- utils.rs
|-- traceroute.rs

3 directories, 15 files
```

`lib.rs` e `main.rs` sono i due crate root files del package:

- `lib.rs`: rappresenta il library crate, ovvero la libreria che fornisce le funzionalità per eseguire traceroute in maniera asincrona.

- `main.rs`: costituisce il binary crate, che utilizza la libreria definita in `lib.rs` per implementare l'utility da linea di comando. Questo crate verrà compilato in un eseguibile consentendo agli utenti di eseguire traceroute direttamente dal terminale, sfruttando le funzionalità fornite dalla libreria.

Il resto dei files sono moduli che vengono importati da `lib.rs` e compilati insieme ad essa. Il modulo `traceroute` è il modulo di più alto livello e contiene una serie di sotto moduli, ognuno dei quali può o non può contenere altri sotto moduli.

2.1 Il modulo `probe`

Il modulo `probe` è il modulo più importante del progetto. Esso contiene:

- cinque strutture dati utilizzate principalmente per contenere dati:
  - `ProbeMethod`: una enumerazione che indica il tipo di probe method da applicare (possibili varianti sono `UDP`, `TCP` o `ICMP`);
  - `ProbeResponse`: una struttura dati costruita a partire da una risposta (ad esempio, una risposta ICMP) a una probe. Contiene:
    - un campo `id` di tipo `ProbeId`, un alias per `String`, usato per identificare sia la richiesta che la risposta,
    - un campo `from_address` che rappresenta l'indirizzo IP del router/host che ha sollevato la risposta.
  - `ProbeResult`: è una struttura simile a `ProbeResponse` ma contiene altri campi come `hostname` e `rtt` (round-trip time);
  - `CompletableProbe`: rappresenta una probe request inviata e in attesa di una risposta. Oltre ai campi `id` per identificare la probe e il campo `from_address`, contiene anche:
    - un campo `sent_at` che tiene traccia di quando è stata inviata la richiesta,
    - un campo `probe_result` che può essere vuoto o meno, a seconda se la risposta è stata ricevuta o no.
  - `ProbeError`: una probe inviata può andare in timeout oppure può verificarsi un errore di I/O. Questa enumerazione ha lo scopo di modellare queste due casistiche.

  La Figura 2 mostra le strutture dati appena elencate[2]:

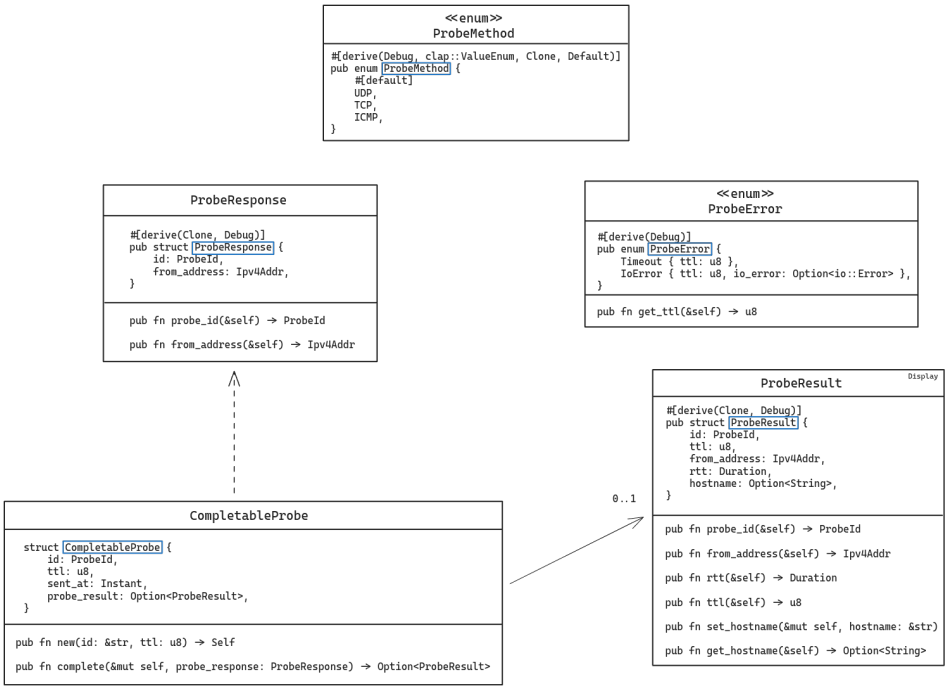

Figura 2: Le strutture dati del modulo `probe.rs`

- quattro sotto moduli: `parser`, `sniffer`, `task`, `generator`

[2]I diagrammi delle classi mostrati in questa documentazione sono in stile UML e utilizzano la sintassi Rust per specificare le struct, le variabili d'istanza e i metodi

2.1.1 Il modulo `parser`

Il modulo `parser` definisce un trait (analogo a un'interfaccia in altri linguaggi) chiamato `ProbeResponseParser`. Tale trait ha lo scopo di fornire un metodo per estrarre un identificativo da una risposta ICMP ricevuta, consentendo la corrispondenza tra la richiesta inviata e la risposta ricevuta. Se l'estrazione dell'identificativo ha successo, il parser ritorna una istanza di tipo `ProbeResponse`.

La Figura 3 mostra il formato di un pacchetto ICMP di tipo `Time Exceeded` e `Destination Unreachable`:

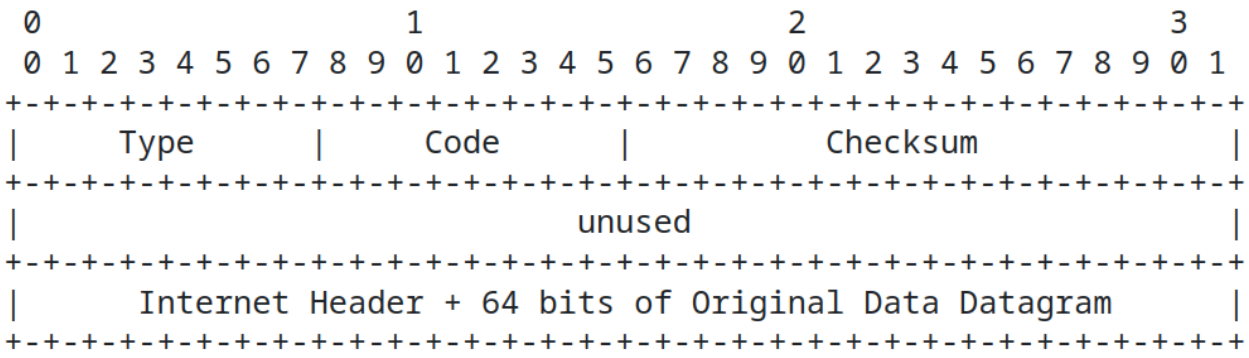

Figura 3: Formato di un ICMP packet di tipo `Time Exceeded` e `Destination Unreachable`

Come descritto nelle specifiche del protocollo ICMP (RFC 792), l'ultima parte di un pacchetto ICMP contiene l'header IP originale e i primi 64 bit dei dati del datagram IP originale. Queste informazioni vengono usate per abbinare la risposta con la richiesta.

Per ogni probe method differente è necessario implementare un corrispondente `ProbeResponseParser`. Attualmente sono implementati tre probe methods, ciascuno con la propria implementazione di `ProbeResponseParser`:

- `UdpProbeResponseParser`: utilizza come identificativo la porta di destinazione. Estrapola la porta di destinazione dall'UDP header contenuto nella risposta ICMP;
- `TcpProbeResponseParser`: utilizza come identificativo il campo `Identification` nell'header IPv4. Estrae il valore del campo `Identification` dall'header IPv4 della probe request originale contenuto nella risposta ICMP;
- `IcmpProbeResponseParser`: la risposta ICMP può essere o un messaggio `ICMP Echo Reply` o un messaggio `ICMP Time Exceeded`. In entrambi i casi, l'identificativo viene costruito concatenando i campi `Identifier` e `Sequence Number` contenuti nel messaggio `ICMP Echo/Echo Reply`, il cui formato è illustrato di seguito (Figura 4):

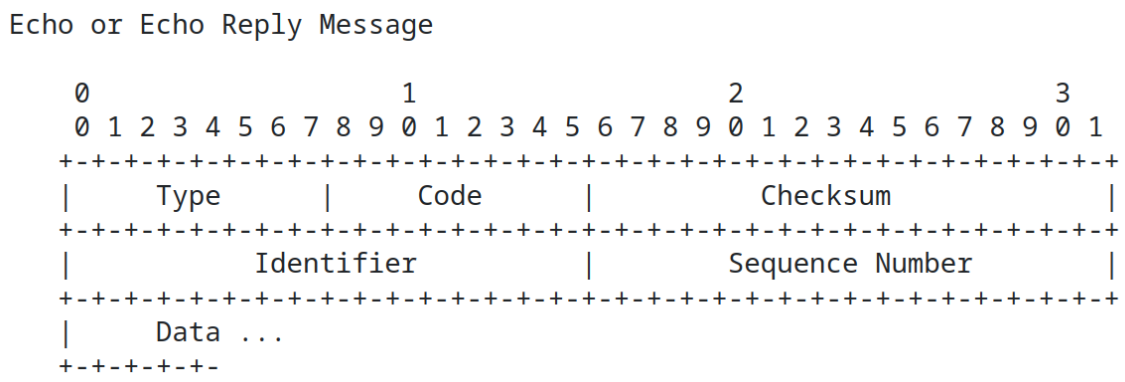

Figura 4: Formato di un ICMP Packet di tipo `Echo/Echo Reply`

Nel caso di una risposta `ICMP Echo Reply`, le informazioni per costruire l'identificativo vengono estratte direttamente da tale messaggio. Nel caso di una risposta `ICMP Time Exceeded`, viene prima estratto il messaggio `ICMP Echo` originale dall'`ICMP Time Exceeded`, e poi costruito l'identificativo.

La Figura 5 mostra il diagramma delle classi del modulo `parser.rs`.

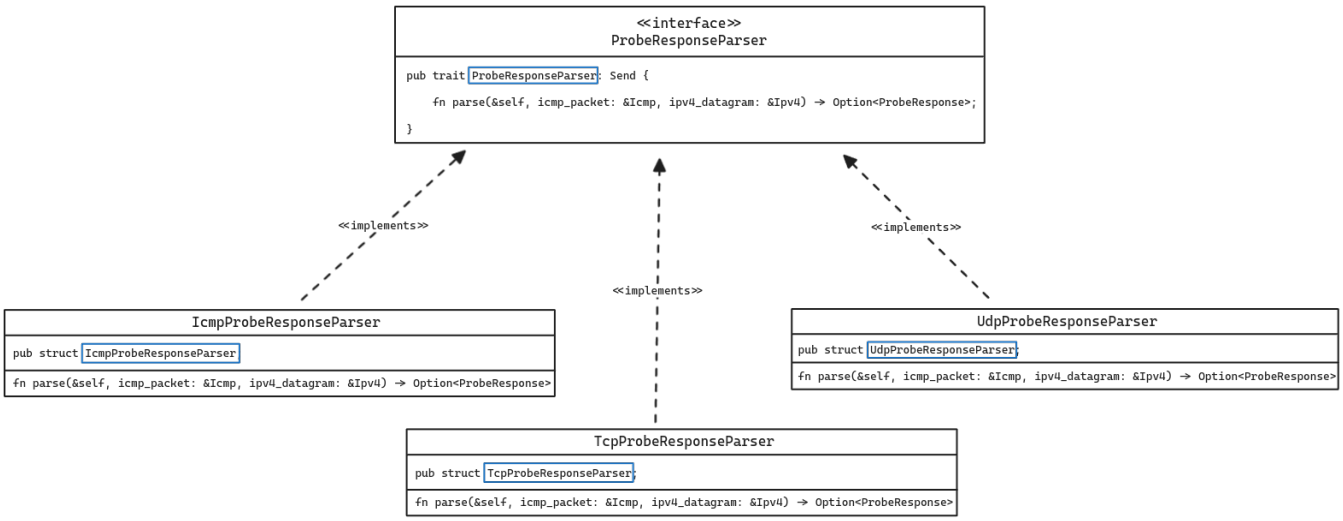

Figura 5: Diagramma delle classi del modulo `parser.rs`

2.1.2 Il modulo `sniffer`

Il modulo `sniffer` definisce il trait `ObservableIcmpSniffer`, il cui compito è intercettare tutte le risposte ICMP in entrata e inviarle ai destinatari in attesa.

L'unica implementazione del trait è la struct `ObservableIcmpProbeResponseSniffer`. Questa struct utilizza un `ProbeResponseParser` per costruire un'istanza di tipo `ProbeResponse` per ogni risposta ICMP intercettata, inviandola poi al destinatario in attesa. Il destinatario si registra come osservatore, aspettando una risposta con un particolare identificativo. La risposta di tipo `ProbeResponse` viene quindi consegnata al destinatario utilizzando l'identificativo costruito e contenuto nella risposta stessa.

La Figura 6 mostra il diagramma delle classi del modulo `sniffer.rs`.

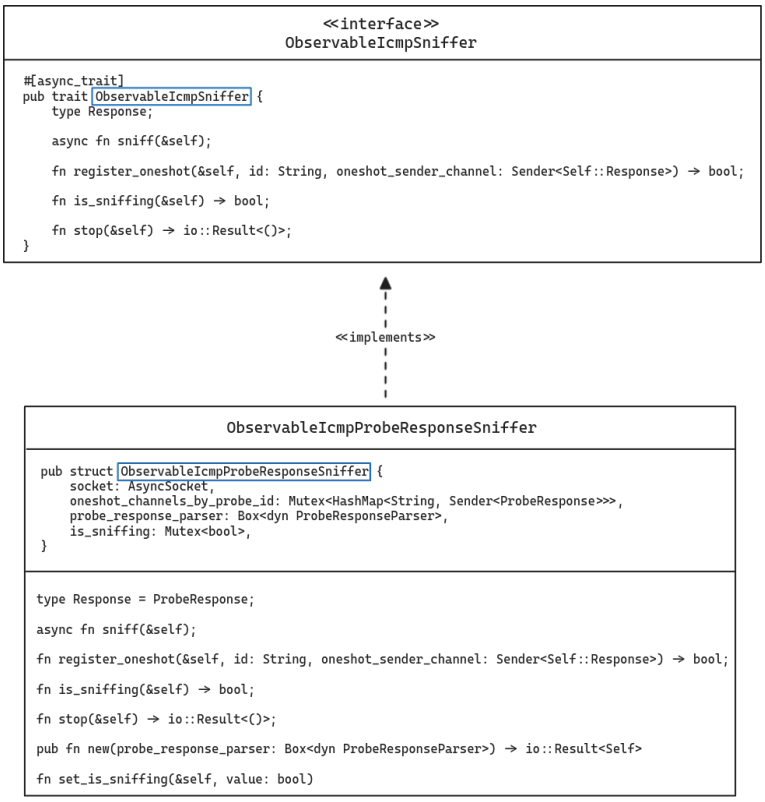

Figura 6: Diagramma delle classi del modulo `sniffer.rs`

2.1.3 Il modulo `task`

Il modulo `task` definisce un trait `ProbeTask`, il cui scopo è inviare uno specifico tipo di probe request e attendere la corrispondente risposta. La richiesta ha un timeout definito: se il timeout scade prima dell'arrivo della risposta, il `ProbeTask` termina e restituisce un `ProbeError::Timeout`. Se invece la risposta arriva entro il timeout, il `ProbeTask` costruisce un `ProbeResult` che riassume il risultato ottenuto. Tutti i `ProbeTask` richiedono un valore per il campo `TTL`, un timeout massimo, un indirizzo di destinazione e un canale di comunicazione in sola lettura per ricevere la risposta ICMP costruita e inviata dall'`ObservableIcmpProbeResponseSniffer`.

Per ogni probe method, esiste un'implementazione specifica del trait `ProbeTask`:

- `UdpProbeTask`: richiede anche una porta di destinazione, utilizzata come identificativo per la probe request inviata e per la corrispondente risposta. Quando una UDP probe request raggiunge l'host di destinazione, quest'ultimo risponderà con un `ICMP Destination Unreachable` (in particolare `Port Unreachable`) se non è in ascolto su quella specifica porta. In caso contrario, la richiesta non riceverà mai una risposta e il task andrà in timeout. La porta di destinazione deve avere quindi un valore improbabile. Inoltre, poiché la porta di destinazione funge da identificativo per la probe (richiesta e risposta), ogni UDP probe request deve avere una porta di destinazione differente;

- `TcpProbeTask`: invia un segmento `TCP SYN` utilizzando una porta di destinazione che è probabilmente aperta sull'host di destinazione (di default la porta 80, associata al protocollo HTTP). Per identificare la probe, viene utilizzato il campo `Identification` nell'header IPv4, quindi è necessario specificare un valore per questo campo (parametro `ip_id`). Nel caso particolare in cui il segmento `TCP SYN` raggiunge l'host di destinazione, se la porta di destinazione sull'host è aperta, l'host di destinazione risponderà con un `TCP SYN-ACK`. Il compito di `TcpProbeTask` è anche quello di catturare questa eventuale risposta. In altre parole, `TcpProbeTask` può ricevere sia un `TCP SYN-ACK` che un messaggio di tipo `ICMP Time Exceeded` come risposta. Pertanto, deve monitorare due canali di comunicazione: il socket utilizzato per inviare il `TCP SYN` e un canale di comunicazione in sola lettura per ricevere eventuali risposte ICMP. Il primo messaggio ricevuto da uno dei due canali determina la conclusione del task, che sarà completato con il risultato corrispondente.

- `IcmpProbeTask`: invia un `ICMP Echo Request` verso la destinazione specificata. Per identificare la probe, utilizza i campi `Identification` e `Sequence Number` presenti nella echo request (vedi Figura 4). L'invio di un `ICMP Echo Request` richiede l'apertura di un raw socket. Tuttavia, piuttosto di aprire un nuovo socket per ogni task creato, il raw socket è gestito da un altro componente e condiviso tramite canali. Il task utilizza questo canale condiviso per inviare le echo requests. La risposta può essere un `ICMP Echo Reply` o un `ICMP Destination Unreachable`, che vengono catturati dall'`ObservableIcmpProbeResponseSniffer` e inviati al task destinatario in attesa.

La Figura 7 mostra il diagramma delle classi del modulo `task.rs`.

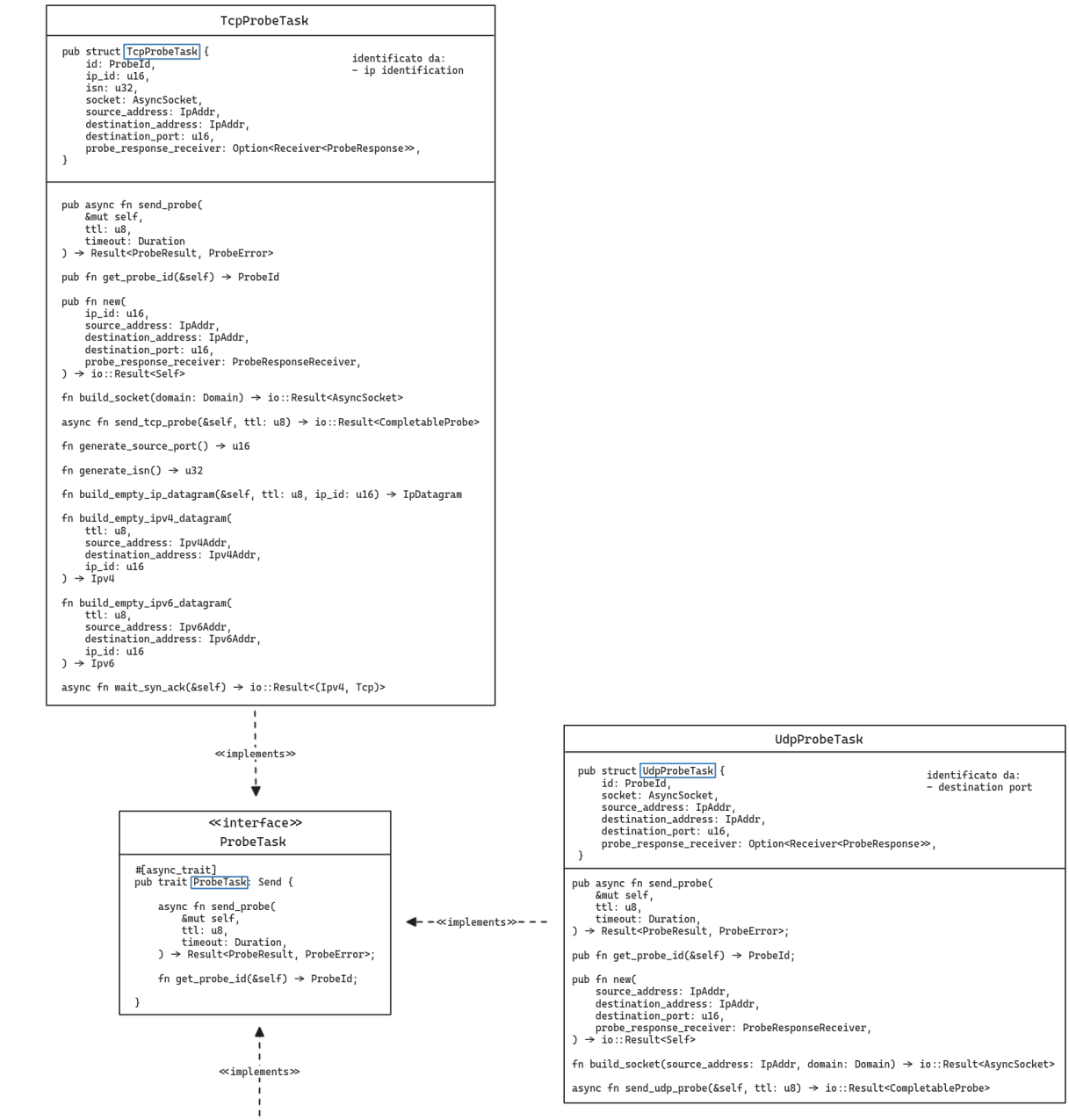

Figura 7: Diagramma delle classi del modulo `task.rs`

2.1.4 Il modulo `generator`

Il modulo `generator.rs` definisce il trait `ProbeTaskGenerator`, progettato per creare istanze di `ProbeTask` incaricate di inviare una probe request e attendere la risposta corrispondente. Il `ProbeTaskGenerator` si occupa di:

- Generare un identificativo univoco per ogni probe
- Registrare tale identificativo con l'`ObservableIcmpSniffer`, garantendo che il `ProbeTask` venga notificato quando viene ricevuta una risposta con lo stesso identificativo
- Fornire tutto il necessario al `ProbeTask` per il suo corretto funzionamento, inclusi uno o più canali sui quali il `ProbeTask` può attendere la risposta

Il metodo `generate_probe_task(...)` restituisce sia il `ProbeTask` creato che il suo identificativo associato. Un `ProbeTask` deve essere istanziato esclusivamente attraverso un `ProbeTaskGenerator`.

La Figura 8 mostra il diagramma delle classi del modulo `generator.rs`.

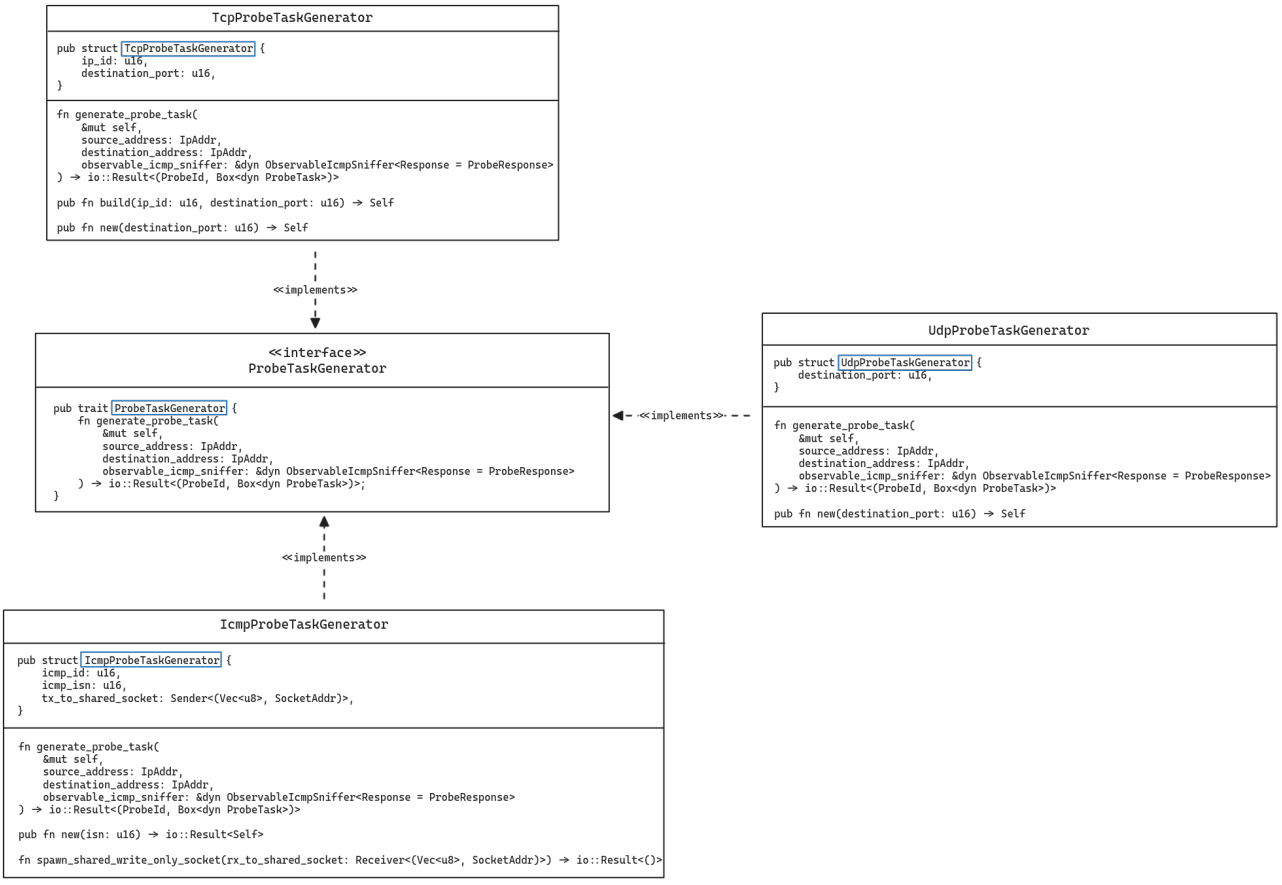

Figura 8: Diagramma delle classi del modulo `generator.rs`

2.2 Il modulo `async_socket`

Il modulo `async_socket.rs` principalmente definisce la struct `AsyncSocket` che permette di usare un socket di qualsiasi tipo usando il modello I/O asincrono. Rust supporta la sintassi `async/await` ma l'esecuzione del codice asincrono, l'I/O e il task spawning sono forniti dagli async runtimes. Il seguente progetto usa `tokio` come async runtime[3].

Il modulo `async_socket` è costituito da tre struct:

- `AsyncSocket`: tokio supporta socket TCP/UDP asincroni. Tuttavia, non fornisce un raw socket asincrono pronto all'uso. I raw sockets sono forniti dal crate `socket2`[4]. Per rendere questi socket asincroni, viene utilizzata la struct `AsyncFd` proveniente dal crate `tokio`. Tale struct permette al tokio reactor di monitorare un oggetto I/O qualsiasi puntato da un descrittore di file Unix e reagire appena questo diventa pronto per la lettura/scrittura[5]. La struct `AsyncSocket` wrappa un `Socket` proveniente dal crate `socket2` in un `AsyncFd`. Questo permette la lettura/scrittura asincrona.

- `SocketWrapper`: questa struct è utilizzata esclusivamente all'interno del modulo. Per poter essere avvolta in un `AsyncFd`, una struct deve implementare il trait `AsRawFd`. Questo trait consente di ottenere il file descriptor di una risorsa I/O, che viene utilizzato dal runtime `tokio` per monitorare il file descriptor. Tuttavia, la struct `Socket` del crate `socket2` non è parte di questo progetto e, di conseguenza, il trait `AsRawFd` (anche esso esterno a questo progetto) non può essere implementato per essa direttamente a causa delle restrizioni di coerenza di Rust[6]. La struct `SocketWrapper` funge da wrapper attorno a `Socket` senza aggiungere funzionalità extra. Il suo unico scopo è permettere l'implementazione del trait `AsRawFd`, consentendo così l'integrazione con il sistema asincrono di Tokio;

- `SharedWriteOnlyAsyncSocket`: questa struct ha lo scopo di creare un socket asincrono e condividerlo tra più thread/task.

La Figura 9 mostra il diagramma delle classi del modulo `async_socket.rs`.

[3] https://tokio.rs/, https://crates.io/crates/tokio

[4] https://crates.io/crates/socket2

[5] Questo viene implementato utilizzando il modello multiplexing I/O attraverso le system call `poll()`, `epoll()`, `kqueue()`, ...

[6] In particolare, questa restrizione fa parte della *orphan rule*, una regola di coerenza che impedisce l'implementazione di trait esterni per tipi esterni

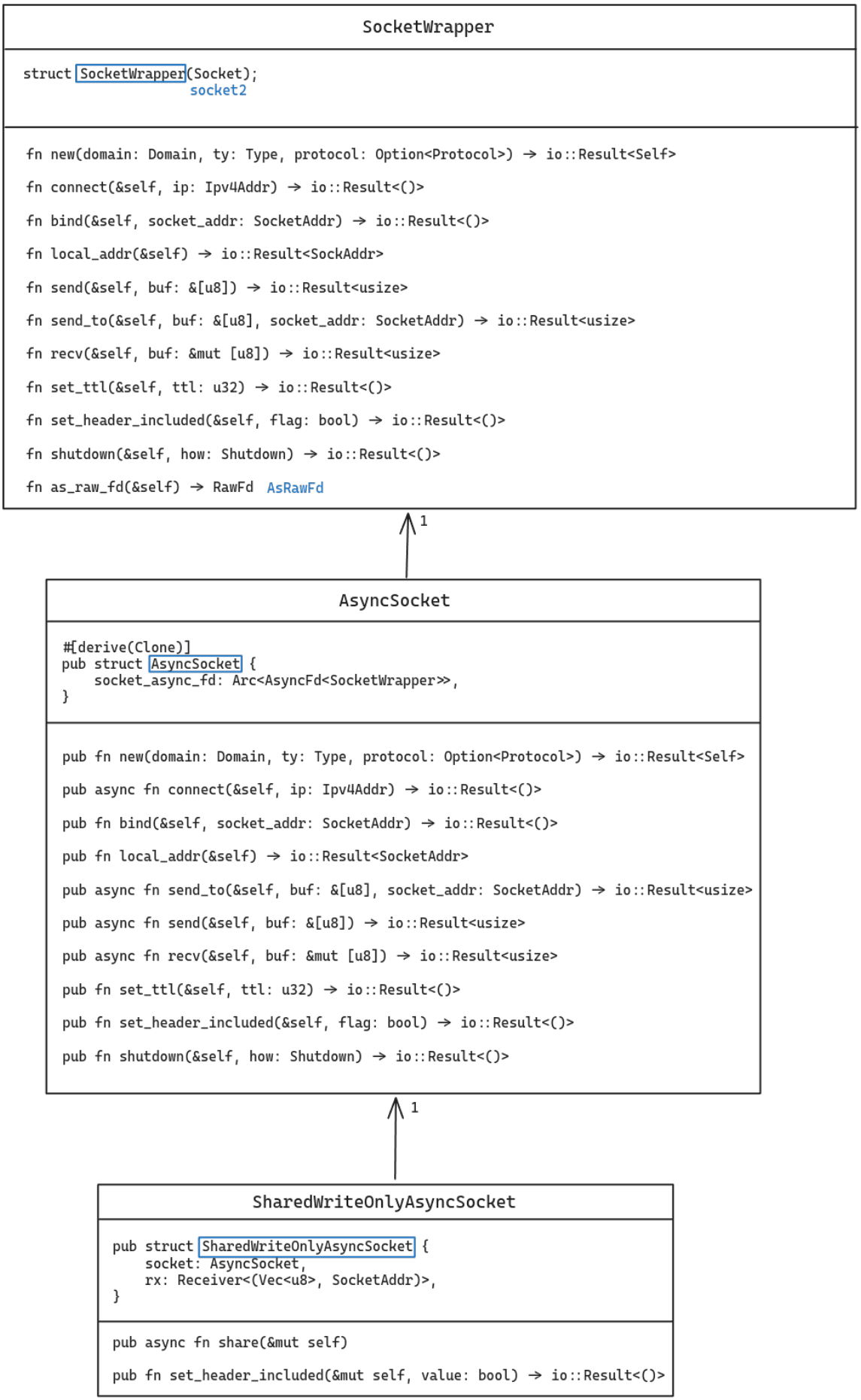

Figura 9: Diagramma delle classi del modulo `async_socket.rs`

2.3 Il modulo `utils`

Il modulo `utils` fornisce una serie di strumenti utilitari suddivisi in tre sottomoduli: `bytes`, `dns` e `packet_utils`:

- `bytes`: questo sottomodulo contiene trait e implementazioni per convertire diverse strutture di pacchetti di rete, come IPv4, UDP, TCP e ICMP, in rappresentazioni byte;
- `dns`: include funzioni asincrone per risolvere nomi di dominio in indirizzi IP e viceversa;
- `packet_utils`: fornisce funzioni per costruire e analizzare vari tipi di pacchetti di rete, inclusi TCP, UDP e ICMP. Include anche strumenti per calcolare checksum e per ottenere indirizzi IP predefiniti dalle interfacce di rete.

2.4 Il modulo `builder`

Il modulo `builder` definisce la struct `TracerouteBuilder`, che semplifica la creazione di un'istanza di `Traceroute`. La costruzione di tutti i componenti necessari per istanziare una struct di tipo `Traceroute` può essere complessa, ma il `TracerouteBuilder` rende questo processo più accessibile fornendo un'API intuitiva e di facile utilizzo.

A seconda del tipo di `ProbeMethod` selezionato, il `TracerouteBuilder`:

- costruisce uno specifico tipo di `ProbeReplyParser`, utilizzato per istanziare un `ObservableIcmpSniffer`
- crea uno specifico tipo di `ProbeTaskGenerator`
- crea un'istanza di `Traceroute` utilizzando i parametri forniti dall'utente (variabili in base al `ProbeMethod` scelto), l'`ObservableIcmpSniffer` e il `ProbeTaskGenerator`.

La Figura 10 mostra il diagramma delle classi del modulo `builder`.

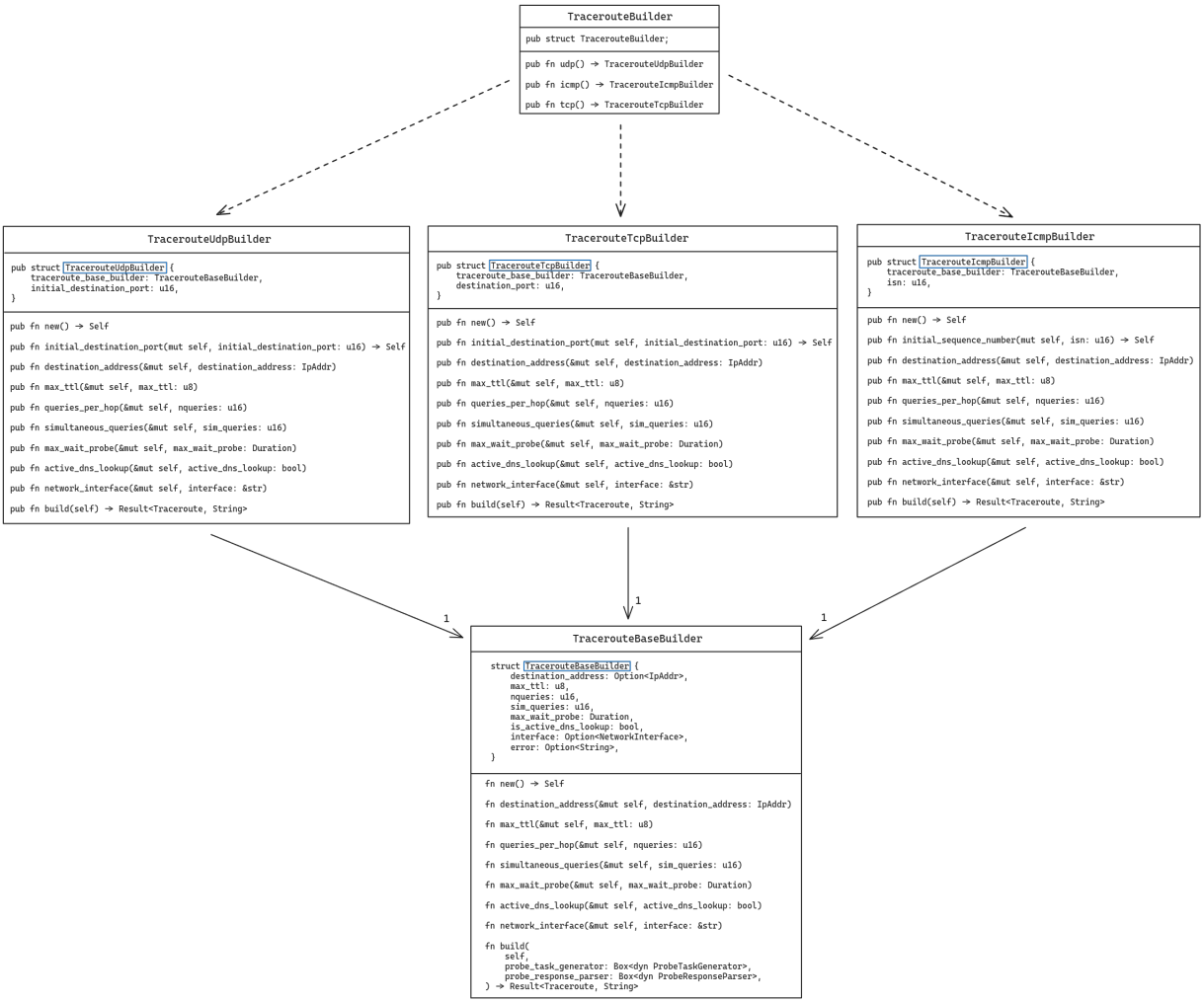

Figura 10: Diagramma delle classi del modulo `builder.rs`

2.5 Il modulo `traceroute`

Il modulo `traceroute` è il modulo principale del progetto e contiene tutti gli altri sottomoduli. Esso definisce la struct `Traceroute`, che utilizza i componenti definiti dagli altri sotto moduli per eseguire traceroute in maniera asincrona verso la destinazione specificata, secondo le modalità stabilite dai parametri forniti.

Una istanza di `Traceroute` deve essere istanziata utilizzando il `TracerouteBuilder`. I parametri che possono essere forniti sono:

- `source_address` (`IpAddr`): l'indirizzo IP di origine;
- `destination_address` (`IpAddr`): l'indirizzo IP di destinazione;
- `max_ttl` (`u8`): il valore massimo di Time To Live (TTL) per i pacchetti inviati, che limita il numero di salti (hops) che i pacchetti possono compiere;
- `nqueries` (`u16`): il numero di probe da inviare per ciascun valore di TTL (per ciascun hop);
- `sim_queries` (`u16`): il numero di probe da inviare simultaneamente;
- `max_wait_probe` (`Duration`): il tempo massimo di attesa per una risposta a una probe request;
- `is_active_dns_lookup` (`bool`): indica se eseguire reverse DNS lookups per gli indirizzi IP delle risposte ricevute;
- `interface` (`NetworkInterface`): l'interfaccia di rete da utilizzare.

La Figura 11 mostra il diagramma delle classi del modulo `traceroute.rs`.

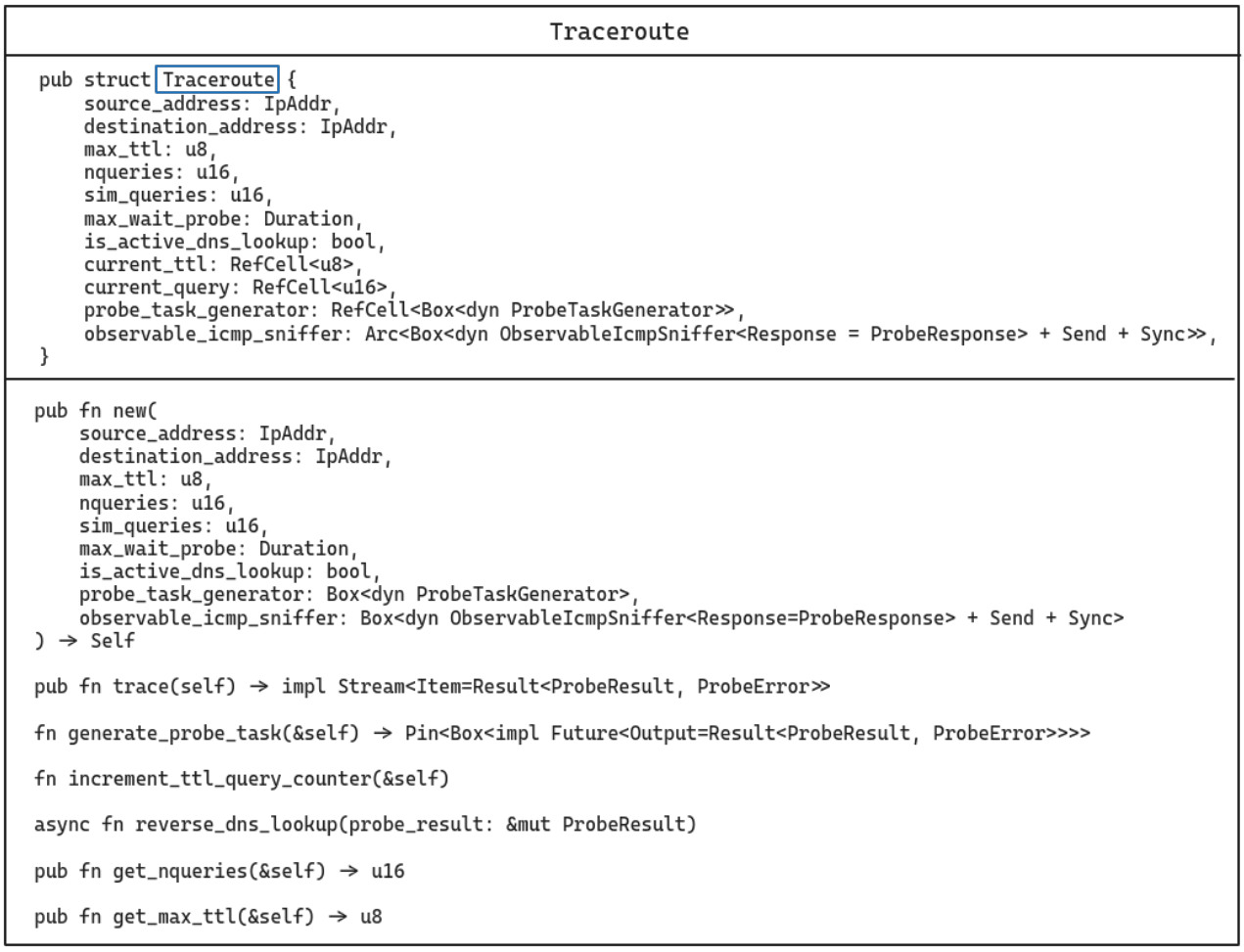

Figura 11: Diagramma delle classi del modulo `traceroute.rs`

2.6 Il modulo `terminal`

Il modulo `terminal` definisce la struct `TracerouteTerminal` con l'obiettivo di eseguire `Traceroute` e visualizzare i risultati in ordine sul terminale. Per creare una istanza di `TracerouteTerminal` è necessario prima istanziare un `Traceroute`. Il metodo `print_trace(...)` di `TracerouteTerminal` avvia traceroute e stampa sullo standard output i risultati in ordine crescente di hop.

La Figura 12 mostra il diagramma delle classi del modulo `terminal`.

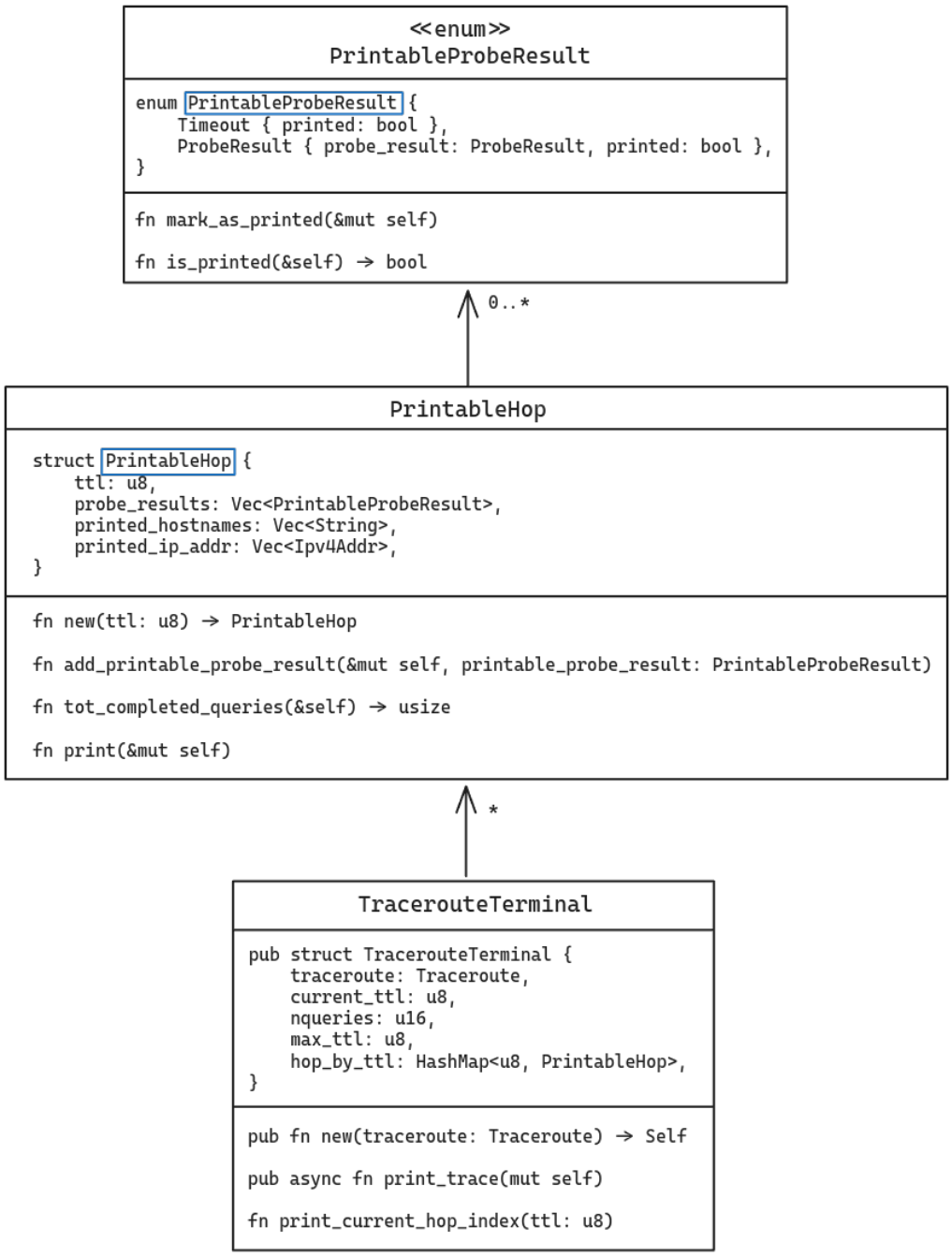

Figura 12: Diagramma delle classi del modulo `terminal.rs`

3 Guida all'utilizzo

3.1 Installazione

I modi più semplici per installare/usare `async-traceroute` sono:

- Installazione con `cargo`
- Utilizzare Docker

3.1.1 Installazione con `cargo`

`cargo` è il build tool e package manager di Rust. Per installare `async-traceroute` con `cargo`, è necessario installare prima Rust[7]. Una volta installato Rust, eseguire i seguenti passaggi:

1. Installa il binary crate `async-traceroute` eseguendo il seguente comando:

```
$ cargo install async-traceroute
```

2. Il binary crate verrà memorizzato nella directory `$HOME/.cargo/bin`. Aggiungere la directory in `$PATH`:

```
$ export PATH=~/.cargo/bin:$PATH
```

3. Esegui `async-traceroute`:

```
$ async-traceroute --help
Async Traceroute library/command-line tool

Usage: traceroute [OPTIONS] <HOST>

Arguments:
  <HOST>

Options:
  ...
```

3.1.2 Docker

1. Clonare il progetto da GitHub:

```
$ git clone https://github.com/vtramo/async-traceroute.git
```

2. Buildare l'immagine tramite `Dockerfile`:

```
$ cd async-traceroute && docker build -t async-traceroute .
```

3. Eseguire `async-traceroute` tramite Docker:

```
$ docker run async-traceroute --help
Async Traceroute library/command-line tool

Usage: traceroute [OPTIONS] <HOST>

Arguments:
  <HOST>

Options:
  ...
```

[7] Installa Rust: https://www.rust-lang.org/tools/install

3.2 Help Output

```
$ async-traceroute --help
Async Traceroute library/command-line tool

Usage:

Arguments:
  <HOST>

Options:
  -m, --max-hops <MAX_HOPS>          Set the max number of hops (max TTL to be reached) [default:
                                     30]
  -q, --queries <QUERIES>            Set the number of probes per each hop [default: 3]
  -w, --wait <WAIT>                  Wait for a probe no more than <WAIT> [default: 3s]
  -N, --sim-queries <SIM_QUERIES>    Set the number of probes to be tried simultaneously [default:
                                     16]
  -P, --probe-method <PROBE_METHOD>  [default: udp] [possible values: udp, tcp, icmp]
  -p, --port <PORT>                  This value changes semantics based on the probe method
                                     selected. It is either initial udp port value for "udp" probe
                                     method (incremented by each probe, default is 33434), or
                                     initial seq for "icmp" probe method (incremented as well,
                                     default from 1), or destination port for "tcp" probe method
                                     (default is 80)
  -n                                 Do not resolve IP addresses to their domain names
  -i, --interface <INTERFACE>        Specify a network interface to operate with
  -h, --help                         Print help
  -V, --version                      Print version
```

3.3 Utilizzare `async-traceroute` come una libreria

1. Creare un nuovo progetto con `cargo`:

   ```
   $ cargo new my-project
   ```

2. Aggiungere `async-traceroute` tra le `dependencies`:

   ```
   $ cargo add async-traceroute
   ```

   Il file `Cargo.toml` dovrebbe essere strutturato come segue:

   ```
   [package]
   name = "my-project"
   version = "0.1.0"
   edition = "2021"

   [dependencies]
   async-traceroute="0.1.2"
   ```

3. Eseguiamo traceroute utilizzando come `ProbeMethod` UDP verso `google.com` tramite il seguente codice (assicurarsi di selezionare la giusta network interface):

   ```rust
   use std::time::Duration;

   use futures::{pin_mut, StreamExt};

   use async_traceroute::{dns_lookup_first_ipv4_addr, TracerouteBuilder};

   #[tokio::main]
   async fn main() -> Result<(), String> {
       let ip_addr = match dns_lookup_first_ipv4_addr("google.com").await {
           None => return Err(String::from("Hostname not resolvable")),
           Some(ip_addr) => ip_addr,
       };
   ```

```
        let traceroute = TracerouteBuilder::udp()
            .destination_address(ip_addr)
            .max_ttl(15)
            .queries_per_hop(3)
            .max_wait_probe(Duration::from_secs(3))
            .simultaneous_queries(16)
            .active_dns_lookup(true)
            .initial_destination_port(33434)
            .network_interface("eth0")
            .build();

        let traceroute_stream = match traceroute {
            Ok(traceroute) => traceroute.trace(),
            Err(error) => return Err(error),
        };

        pin_mut!(traceroute_stream);
        while let Some(probe_result) = traceroute_stream.next().await {
            println!("{:?}", probe_result);
        }

        Ok(())
    }
```

3.4 Esempi di utilizzo

```
# UDP Probe Method
async-traceroute google.com

# TCP Probe Method
async-traceroute -P tcp google.com

# ICMP Probe Method
async-traceroute -P icmp google.com

# TCP Probe Method usando 443 come porta di destinazione
async-traceroute -P tcp -p 443 google.com

# ICMP Probe Method usando l'interfaccia di rete wlp3s0 e un sequence number iniziale uguale a 42
async-traceroute -P icmp -i wlp3s0 -p 42 google.com

# UDP Probe Method con un TTL massimo di 10 e un timeout per ogni probe di 1 secondo
# e con porta iniziale uguale a 35000
async-traceroute -m 10 -w 1s -p 35000 google.com

# UDP Probe Method con timeout di 1 secondo inviando una sola probe per ogni hop
# aspettando la risposta prima di inviare un'altra
async-traceroute -q 1 -N 1 -w 1s google.com

# Non eseguire reverse dns lookups
async-traceroute -n google.com
```

3.5 Repository GitHub

https://github.com/vtramo/async-traceroute

3.6 Limitazioni

- Non compatibile con Windows

- Richiede i privilegi di amministratore per essere eseguito
- Nessun supporto IPv6