Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE
Corso di Laurea in Informatica

Studio e implementazione di strumenti di domain name resolution per l'Internet of Threads

Relatore:
Chiar.mo Prof.
Renzo Davoli

Presentata da:
Federico De Marchi

Sessione I
Anno Accademico 2018/2019

People think that computer science is the art of geniuses
but the actual reality is the opposite,
just many people doing things that build on eachother,
like a wall of mini stones.
Donald Knuth

Indice

Introduzione **1**

1 Stato dell'Arte **3**

1.1 Internet Protocol, Version 6 3
1.1.1 Indirizzamento IPv6 3
1.1.2 Generazione di indirizzi IPv6 4
1.1.3 Indirizzi Hash IPv6 basati su FQDN 6
1.2 Domain Name System 8
1.2.1 Estensioni per DNS 8
1.2.2 Trasmissione DNS attraverso TCP 8
1.2.3 hashdns 9
1.3 IoTh, l'Internet of Threads 10
1.3.1 Processi come nodi di Internet 10
1.3.2 libioth 11
1.3.3 One Time IP Address 12

2 Sviluppo di VdeDNS **15**

2.1 Analisi progettuale 15
2.1.1 Finalità 15
2.1.2 Scelte implementative 16
2.2 Implementazione 19
2.2.1 Struttura di VdeDNS 19
2.2.2 Strutture dati principali 20
2.2.3 DNS Parser 23
2.2.4 Forwarding di richieste 25
2.2.5 Implementazione UDP 26
2.2.6 Implementazione TCP 27
2.3 Sperimentazione 30
2.3.1 Setup e risoluzione indirizzi 30
2.3.2 Performance di VdeDNS 32

3 Tool di supporto per OTIP **35**

3.1 Finalità e descrizione 35
3.1.1 OTIP TCP Proxy 35
3.1.2 OTIP UDP Proxy e Tunnel 36
3.2 Implementazione 36

3.2.1 OTIP TCP Proxy 37
3.2.2 OTIP UDP Proxy 37
3.2.3 OTIP UDP Tunnel 39
3.3 Sperimentazione 40
3.3.1 SSH 40
3.3.2 Web Server 41
3.3.3 Echo server UDP 43

Conclusione e sviluppi futuri **47**

Appendices **51**

A OTIP Web Server Tutorial **53**
A.1 Proxy Setup 53
A.1.1 Proxy on kernel stack 53
A.2 Enabled server setup 54
A.3 Access using vdedns 54

Introduzione

La rapida e continua espansione di Internet, sia nel numero di dispositivi connessi sia nel suo sviluppo infrastrutturale, continua a rendere il suo relativo campo di ricerca sempre aperto a nuovi esperimenti e prototipi.

Facendo uso dell'attuale stato dell'arte dei protocolli di rete IPv6 (Internet Protocol Version 6) e DNS (Domain Name System) e degli ultimi sviluppi inerente alla virtualizzazione di stack di rete a livello di processo, si vuole costruire un aggiornamento su precedenti proposte [5, 6] di utilizzo di indirizzi IPv6 generati tramite funzioni di hashing. Queste proposte d'uso per gli indirizzi IPv6 generati da funzioni hash riguardano due differenti temi: la semplificazione di configurazione di reti di host tramite nomi di dominio e l'aggiunta di un layer di protezione a server a uso privato. Per semplificare la configurazione di host associati a un dominio si possono generare proceduralmente indirizzi IPv6 basati sul nome di sottodominio degli host stessi, mentre un server può limitare gli accessi tramite uso di indirizzi IPv6 temporanei, generati da parametri conosciuti solo dalle entità autorizzate.

L'uso di funzioni hash per produrre indirizzi IP non è di per sé una novità, ma l'uso corrente della tecnica riguarda principalmente il mantenimento della privacy nel processo di autoconfigurazione di interfacce di rete. Questi sistemi sono consolidati e già standardizzati dalla Internet Engineering Task Force e sono solitamente messi in uso per prevenire il tracciamento di sistemi mobili (computer portatili, smartphone e simili) autoconfiguranti negli spostamenti di rete IPv6.

Le proposte alternative sopracitate sono invece tuttora sperimentali e i prototipi relativi sono solo proof-of-concept, sviluppati con tool e idee di uno stato dell'arte ormai parzialmente datato. I tool sviluppati a oggi nell'ambito degli indirizzi basati su nomi di dominio sono già in parte funzionali e pronti all'uso, ma i tool sviluppati per l'uso di indirizzi temporanei esistono solo in stato dimostrativo. Nel dettaglio, entrambi presentano già risolutori DNS disponibili, che sono tuttavia separati, hanno opzioni di configurazione limitate e/o richiedono un setup non banale e sono quindi poco idonei a un uso pratico. Solo gli indirizzi basati su nomi di dominio hanno però strumenti di supporto sufficienti per poter costruire sistemi completi, mentre lo stato attuale del supporto per indirizzi temporanei non ne permette ancora un uso reale.

Lo scopo di questo lavoro è effettuare un'analisi dell'operato precedente sul tema dell'utilizzo di indirizzi hash IPv6, valutando le soluzioni finora proposte allo scopo di trovare i loro punti di successo e di carenza, per poi valutarne la reale attuabilità ed eventuali limiti costruendo nuovi prototipi e sperimentandone il funzionamento in congiunzione con altri programmi in ambienti reali e/o simulati.

Il lavoro si apre con una panoramica del presente stato dell'arte del protocollo IPv6, soffermandosi in particolare sulla generazione di indirizzi hash, dei requisiti e feature

dei sistemi DNS più recenti e infine sulla virtualizzazione di stack di rete a livello di processo nell'Internet of Threads. La parte centrale del lavoro riguarda l'implementazione di VdeDNS, un proxy DNS dedicato alla risoluzione di entrambe le tipologie dei suddetti indirizzi hash IPv6, creato allo scopo di rendere facilmente accessibili le loro feature, senza necessità di configurazioni complesse. Il proseguimento si interessa invece dell'implementazione di prototipi sperimentali che permettano di costruire sistemi protetti dal meccanismo di indirizzi IPv6 temporanei, facendo uso dei recenti sviluppi sulla virtualizzazione di reti. Entrambe le parti implementative sono seguite da esperimenti per verificare il funzionamento e misurare l'efficacia degli strumenti costruiti.

Capitolo 1

Stato dell'Arte

1.1 Internet Protocol, Version 6

Le implementazioni proposte nell'elaborato sono interamente basate sull'uso dell'indirizzamento IPv6 e sono incentrate sullo sfruttarne il potenziale dato dal suo incredibilmente esteso spazio di indirizzi. L'adozione globale di IPv6 è in crescita costante (come indica Google [13]) ed è ormai giustificabile la creazione di applicazioni che ne facciano uso dedicato, anche allo scopo di incoraggiare una spinta verso un'ulteriore crescita con l'introduzione di nuove opportunità offerte dalla tecnologia.

1.1.1 Indirizzamento IPv6

La principale caratteristica di IPv6 è la sua altissima scalabilità data dai suoi 2^{128} indirizzi unici, essendo il protocollo progettato perché sia possibile assegnare a ogni singola interfaccia di rete all'interno di un nodo di Internet un indirizzo pubblico individuale. Questa caratteristica dovrebbe idealmente essere accessibile a chiunque, dato che con l'RFC6177 [19] è fortemente suggerito che perfino alle abitazioni private vengano assegnati indirizzi IPv6 almeno con prefisso /64 o inferiore per essere in grado di indirizzare univocamente e liberamente i propri dispositivi sulla rete pubblica. In caso di espressa necessità non dovrebbe essere difficile fare richiesta per un indirizzo /48, anche come ente privato.

I 128 bit di un indirizzo IPv6 sono rappresentati come una catena di 32 valori esadecimali, separati in 8 blocchi da 4 valori ciascuno. Sono definite tre tipologie di indirizzi IPv6:

Unicast Identifica una singola interfaccia

Anycast Identifica un insieme di interfacce, un pacchetto è recapitato a una di esse

Multicast Identifica un insieme di interfacce, un pacchetto è recapitato a ognuna

Il broadcast di IPv4 è assente perché sostituito dal multicast, che può essere considerato una sua generalizzazione. Mentre un indirizzo multicast è identificato dai primi 8 bit dell'indirizzo messi a 1, un indirizzo anycast è indistinguibile da un indirizzo unicast. Un indirizzo unicast diventa invece anycast semplicemente nel momento in cui è assegnato

a più interfacce, l'unico requisito per il funzionamento è che il nodo a cui è assegnato sia configurato di conseguenza.

Un indirizzo unicast pubblico globale è generalmente composto da:

- N bit di prefisso globale di routing
- $64 - N$ bit di identificazione della sottorete
- 64 bit di identificazione dell'interfaccia di rete

In un indirizzo unicast, a eccezione degli indirizzi che iniziano con i primi 3 bit a 0, l'identificativo dell'interfaccia è sempre di 64 bit e deve essere unico all'interno della sottorete. Questo tipo di indirizzi si costruiscono aderendo al formato *EUI-64 modificato*. In questo formato, un identificativo ha scope universale quando è derivato da un token universale, come nel caso di un identificatore di un'interfaccia MAC di rete fisica assegnato direttamente dall'IEEE attraverso il suo produttore, altrimenti ha scope locale. Un indirizzo assegnato da un amministratore di sistema è considerato locale e deve avere conseguentemente il bit universale (7° bit dell'interfaccia) messo a 0, laddove un indirizzo con scope universale dovrebbe averlo a 1. [9]

È evidente come il punto focale dell'organizzazione degli indirizzi di IPv6 non sia più il dover allocare in maniera ottimale lo spazio, ma diventi il poter progettare sottoreti ampie e appropriate ai bisogni senza essere costretti a ricorrere a tecniche di conservazione dello spazio di indirizzamento (dato che uno spazio di 64 bit permette di allocare tutti gli indirizzi possibilmente necessari). [20, 29] L'abbondanza di indirizzi rende inoltre obsoleto il sistema di Network Address Translation (NAT) usato per associare più dispositivi a un singolo indirizzo IP pubblico, che ha l'effetto collaterale di rendere impossibile la connettività end-to-end dei beneficiari del sistema con l'esterno. Un dispositivo sotto una rete NAT è normalmente in grado di accedere a Internet, ma non può essere direttamente contattato dall'esterno senza dipendere dal dispositivo abilitante, che si interpone nella comunicazione diretta. Il sistema di NAT limita fortemente l'accessibilità e l'indipendenza dei dispositivi e aggiunge potenziali rischi nei processi di autenticazione, portando ad esempio forti svantaggi per reti di dispositivi della categoria dell'Internet of Things (IoT). [29, 30] Un altro vantaggio portato dalla maggiore libertà nel costruire sottoreti è un miglioramento dell'efficienza del routing. Mentre la natura limitata degli indirizzi IPv4 non lo permetteva, la flessibilità dei più lunghi indirizzi IPv6 permette di aggregare facilmente gli indirizzi per gerarchie di reti globali, provider Internet, aree geografiche o simili, limitando le dimensioni delle tabelle di indirizzamento dei principali nodi di routing e migliorandone in generale l'efficienza. [20, 30]

1.1.2 Generazione di indirizzi IPv6

L'ingente spazio di allocazione di IPv6 permette di costruire nuovi metodi di generazione di indirizzi pubblici per interfacce di rete senza doversi eccessivamente preoccupare di generare conflitti all'interno di una stessa sottorete.

Il più comune esempio di questi metodi è lo Stateless Address Auto-configuration (SLAAC), che permette a un dispositivo appena entrato in una rete di potersi autonomamente generare un indirizzo IPv6 pubblico, senza dover dipendere da un servizio di Dynamic Host Configuration Protocol (DHCP). La generazione avviene semplicemente

unendo un prefisso di rete ottenuto tramite advertisement dal router a un suffisso di interfaccia generato dall'interfaccia di rete stessa. Questo indirizzo viene poi verificato dall'algoritmo di Duplicate Address Detection (DAD), che verifica attraverso informazioni trasmesse dal router (tramite messaggi di Neighbor Advertisement) che l'indirizzo sia unico. Nel raro caso l'indirizzo non sia unico, questo non viene assegnato ed è richiesta una configurazione manuale da parte dell'amministratore di sistema. [18]

Da tale tipo di approccio possono purtroppo derivare problemi relativi alla privacy. Un'interfaccia di rete si occupa autonomamente di generare la propria parte di indirizzo, tipicamente utilizzando il formato EUI-64 derivato da token universale (come l'indirizzo MAC unico, assegnato dal produttore) quando si tratti di un'interfaccia fisica o in alternativa generandone una temporanea casualmente con scope locale. Mentre nel secondo caso l'instabilità del suffisso di interfaccia non crea problemi (al di fuori del complicare la gestione di una rete), nel primo caso l'identificatore di interfaccia dell'indirizzo resterà sempre invariato e direttamente riconducibile al dispositivo. Il rischio diventa particolarmente evidente per i dispositivi mobili come smartphone, tablet e dispostivi IoT, poiché diventa possibile tracciare gli spostamenti di una persona attraverso le differenti reti a cui questi vengono connessi, ma anche per computer portatili, per esempio nel cambio di rete da casa al luogo di lavoro. [15]

In risposta a questa problematica viene l'RFC4941 [17], che propone di sostituire la parte statica di interfaccia di indirizzi generati attraverso SLAAC con un valore casuale, generando così indirizzi temporanei. Un primo approccio proposto utilizza un valore di history H di 64 bit da tenere in memoria e generato inizialmente in modo casuale e un identificativo I di 64 bit generato dall'interfaccia di rete come in precedenza. Questi valori sono uniti in V passando attraverso una funzione hash (ad esempio MD5, come consigliato), i primi 64 bit di output sono presi come indirizzo e i restanti 64 bit diventano il nuovo valore di H ($V = h(H||I)$). L'approccio alternativo non avendo a disposizione una forma di storage è semplicemente generare il valore casualmente.

Per quanto la soluzione adottata risolva il problema, l'uso di indirizzi temporanei può rendere molto difficile l'amministrazione di una rete: l'event logging, la risoluzione di problemi, il controllo degli accessi, i controlli di qualità del servizio... diventano eccessivamente complicati (se non quasi impossibili) da gestire.[12] La proposta dell'RFC7217 [12] è un metodo per generare identificatori di interfaccia per SLAAC che restino stabili all'interno di una stessa sottorete, ma che cambino con il cambiare di questa. Usando una funzione pseudocasuale F difficilmente reversibile, si calcola il valore dell'identificativo di interfaccia V come:

$$V = F(P||I||N||D||K)$$

dove P è il prefisso di rete ottenuto tramite router advertisement, I è il prefisso generato dall'interfaccia (come ad esempio quello statico generato come EUI-64), N è un identificatore di sottorete, D è un contatore di conflitti DAD e K è una chiave segreta generata dall'host. Se l'indirizzo ottenuto unendo prefisso di rete a V passa il controllo DAD, allora è valido, altrimenti si aumenta il contatore D e si riprova.

Questi erano esempi di forme di costruzione di indirizzi IPv6 validi per permettere ai dispositivi di autoconfigurarsi proteggendone al contempo la privacy, a patto di utilizzare sistemi di generazione di numeri casuali sicuri. La differenza principale tra gli ultimi due metodi è il livello di privacy che si vuole ottenere: il primo metodo genera indirizzi temporanei e mai riconducibili al dispositivo anche quando registrato più volte sotto la

stessa rete, il secondo rende gli indirizzi non riconducibili al dispositivo solo nei cambi di rete, ma consente di identificare univocamente il dispositivo quando collegato alla stessa. [15] Gli indirizzi generati da sistemi di questo genere sono tutti considerati amministrati localmente, di conseguenza dovrebbe essere premura dell'algoritmo di generazione di mettere il bit universale a 0.

1.1.3 Indirizzi Hash IPv6 basati su FQDN

I precedenti metodi di autoconfigurazione permettono a un'interfaccia di rete di configurarsi autonomamente senza alcun intervento umano, ma, anche quando stabili (i.e. RFC7217), lasciano la mappatura della rete nelle mani dell'amministratore di sistema. Questo risulta particolarmente problematico nella fase di configurazione del Domain Name System (DNS) locale, dove l'associazione tra nome e indirizzo degli host viene eseguita manualmente ed è quindi suscettibile all'errore umano.

Un'ulteriore proposta [5] è una forma di autoconfigurazione basata sull'uso di Fully Qualified Domain Name (FQDN) per generare indirizzi IPv6 basati sull'hashing di un FQDN associato a un host e il prefisso della rete locale su cui questo si trova. L'obiettivo di questo nuovo metodo è quello di permettere una semplificata e più chiara gestione di una rete agli amministratori di sistema, che semplicemente assegnando un FQDN a un host lo forniscono di un indirizzo e di un dominio per la risoluzione DNS. Il sistema è idoneo a un utilizzo basato su una configurazione sia *stateful*, quindi gestita centralmente da un server DHCP in possesso di una lista di FQDN dei nodi di rete, che assegna agli host richiedenti il loro indirizzo IPv6 dopo aver applicato l'algoritmo di hashing, sia *stateless*, quindi, come visto in precedenza, che lascia all'host il compito di autoconfigurarsi.

L'implementazione di un sistema basato su configurazione stateful si può banalmente ottenere computando le tavole del server DHCP e del server DNS a partire da una lista di tutti gli host e dei loro FQDN. A questo punto un host può richiedere il suo indirizzo mandando una richiesta DHCP con il suo FQDN come identificatore. La limitazione di questa implementazione è la sua staticità, dato che è necessario conoscere a priori tutti gli host della rete per computare le tavole degli indirizzi, che andrebbero inoltre ricompilate ogni volta che si intendano modificare o aggiungere host. D'altra parte l'implementazione basata su un sistema stateless, per quanto permetta a un host di autoconfigurarsi nella rete dinamicamente, richiede dei tool più specifici: un programma per eseguire l'autoconfigurazione sull'host, che possa computare l'indirizzo a partire dal prefisso di rete (ottenuto tramite router advertisement) e dal proprio FQDN, e un server DNS che sia in grado di elaborare risposte a query per indirizzi o domini hash.

Nel concreto, l'implementazione proposta per una rete a configurazione stateless prevede l'uso di un router che faccia advertisement del prefisso di rete, di un server DNS in grado di fare risoluzioni dirette e inverse dei domini hash e di host che sappiano autoconfigurarsi sulla base del loro FQDN. Le specifiche per un DNS specializzato non sono ovvie. Mentre per la risoluzione diretta di un dominio è sufficiente applicare l'algoritmo di hashing su prefisso/i di rete e FQDN per ottenere il record AAAA di risposta, questo stesso algoritmo non è invertibile nel caso di una richiesta di risoluzione inversa. La soluzione proposta consiste nel mantenere in cache una tavola di corrispondenze tra domini risolti e i loro indirizzi, ma un'implementazione naif metterebbe facilmente a rischio la sicurezza del DNS, che potrebbe essere inondato di richieste fittizie da un attaccante

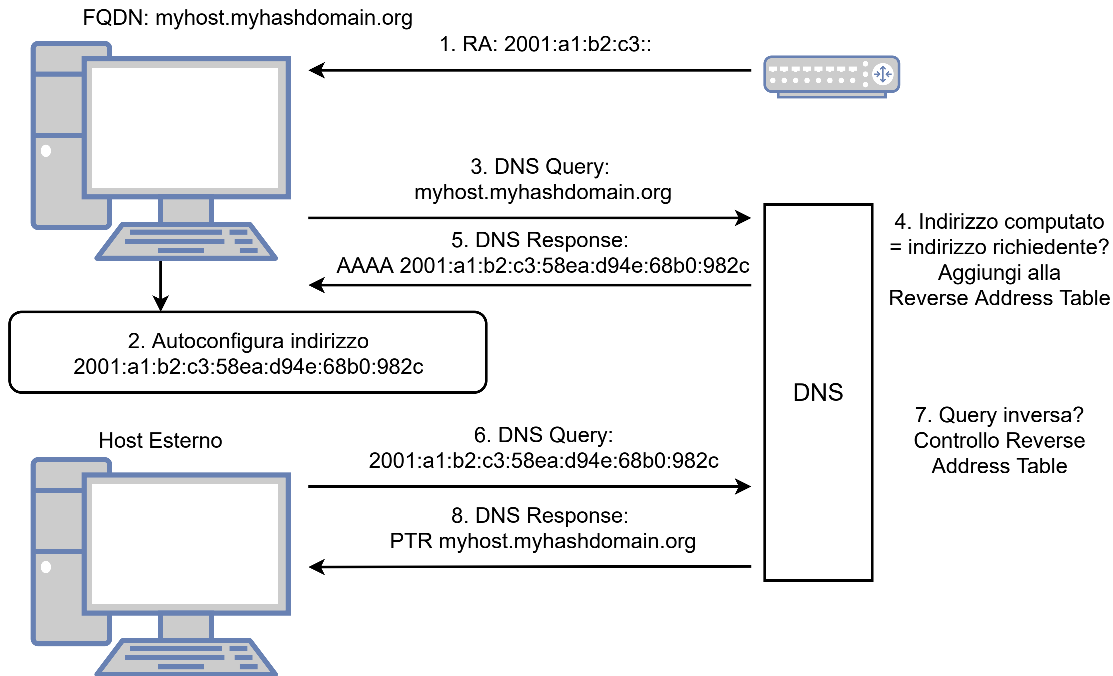


Figura 1.1: Esempio dinamica di rete

provocando overflow di memoria o delay nel servizio. Una possibile implementazione consigliata per tale sistema è di memorizzare nella cache una coppia dominio-indirizzo solo nel caso la query provenga dallo stesso indirizzo che viene risolto, in questo modo un host può periodicamente chiedere una risoluzione del proprio FQDN per mantenere la tavola aggiornata. Le specifiche suggerite per l'algoritmo di generazione sono minimali, quindi le possibili scelte implementative sono innumerevoli. Allo stato attuale e nell'esempio proposto, l'indirizzo è calcolato come:

$$PREFX + [SUFX \oplus MD5SUM(FQDN)[0:63]]$$

dove PREFX è il prefisso di 64 bit dell'indirizzo base della rete, SUFX è il suffisso di 64 bit dello stesso, + è l'operatore di append e sono presi in considerazione solo i primi 64 bit della funzione di hashing. [22]

Un possibile esempio di funzionamento base (figura 1.1) è il caso in cui un nuovo host con un FQDN assegnato si connetta alla rete. Appena entrato sollecita il router e riceve l'indirizzo base di rete, che gli permette di generarsi autonomamente un indirizzo IPv6. A questo punto, per rendersi rintracciabile, richiede al server DNS di risolvergli il suo stesso dominio: il DNS si accorge che l'indirizzo risolto corrisponde all'indirizzo del richiedente e di conseguenza aggiunge alla cache (Reverse Address Table) la coppia $(FQDN, indirizzo)$. Questa operazione va ripetuta periodicamente allo scadere dell'elemento in cache. Dopo questo procedimento di configurazione iniziale, un qualunque host, anche esterno alla rete, può richiedere al DNS la risoluzione inversa dell'indirizzo del nuovo host locale.

Come si può osservare, l'autoconfigurazione tramite FQDN permette di rendere trasparente la configurazione di una rete rappresentando gli host come nomi gerarchici. Questo tipo di approccio rende senz'altro più facile la manutenzione e l'amministrazione di un sistema, sostituendo l'uso di indirizzi IPv6 puri con più naturali identificatori.

1.2 Domain Name System

Il Domain Name System (DNS) è fin dalla sua introduzione uno degli elementi fondamentali per il funzionamento di Internet. Il DNS permette di associare a risorse in rete un proprio nome, allo scopo di poterle facilmente recuperare attraverso un sistema gerarchico e decentralizzato, utilizzando linguaggio naturale invece che indirizzi di macchine. Uno dei temi principali dell'elaborato è lo studio di come costruire un prototipo conforme agli standard DNS evoluti nel corso dell'ultimo decennio.

1.2.1 Estensioni per DNS

Lo standard dettato dall'originale RFC1035 [16] per l'implementazione di DNS non ha mai visto variazioni nel possibile numero di campi disponibili nell'header. eon la recente evoluzione ed espansione dei Content Delivery Network (CDN) si è reso necessario ideare una soluzione che permetta di aumentare le capacità e permettere la crescita del protocollo mantenendolo retrocompatibile con le specifiche iniziali. Extensions Mechanism For DNS (EDNS) [3] è una specifica per abilitare il protocollo DNS standard all'aggiunta di estensioni facendo potenzialmente uso di label ancora inutilizzate nell'header DNS e aggiungendo nuovi Resource Record (RR).

EDNS inserisce informazioni aggiuntive nei pacchetti DNS utilizzando cosiddetti "pseudo-RR", aggiunti nella sezione dati di un pacchetto marcati come un RR di tipo OPT. Un RR OPT ha un header fisso, con opzioni di metadata standard del protocollo e alcune opzioni aggiuntive ritenute utili, e una parte variabile che può contenere dati arbitrari marcati da un codice operazionale. Affinché un server DNS sia conforme a una specifica estensione DNS, è necessario sia in grado di leggere record OPT da una query e rispondere in accordanza con il protocollo dell'estensione con un record OPT dello stesso tipo. Questo implica che un server possa scegliere di implementare solo una certa porzione delle estensioni esistenti senza causare problemi di incompatibilità alla radice dello standard DNS: nel caso un client rilevi la mancanza di risposta all'estensione richiesta, può riprovare mandando una query standard non estesa.

Con l'espansione del protocollo DNS e con l'aggiunta di EDNS diventa più comune che un pacchetto UDP superi la soglia di 512 byte imposta dallo standard iniziale. Oltre al ritentare una query attraverso TCP come suggerito dallo standard, EDNS introduce un'opzione per dare la disponibilità ad accettare risposte fino a una nuova specificata soglia. È ovviamente consigliato non mandare advertisement di una soglia al limite architetturale del protocollo (64 KB), per evitare sprechi di memoria ai livelli di elaborazione dei pacchetti di rete nei sistemi.

1.2.2 Trasmissione DNS attraverso TCP

La grande maggioranza delle transazioni DNS avviene tramite UDP, ma la frequenza di quelle eseguite tramite TCP è aumentata in modo rilevante nell'ultimo decennio. [2] Con la crescita dei client e server aderenti ai protocolli di DNSSEC e con l'aumento di popolarità di IPv6 è sempre più comune che si esegua una richiesta attraverso TCP per pacchetti di dimensione superiore ai 512 byte di soglia standard UDP. Non solo, TCP

rende più sicure le transazioni DNS fornendo protezione da attacchi relativi all'address spoofing e può fornire un layer di crittografia con l'uso di DNS-over-TLS. [10]

L'RFC7766 [10] fornisce le linee guida standard per l'implementazione del trasporto DNS attraverso TCP. Il principale uso del trasporto TCP per DNS è come meccanismo di fallback dove un pacchetto di risposta UDP possa eccedere le dimensioni limite: in questo caso la procedura standard è di mandare il pacchetto di risposta troncato accendendo il cosiddetto truncation bit (flag TC) nell'header, in modo che il client capisca di dover ritentare la richiesta con TCP. Le nuove linee guida suggeriscono che TCP possa essere utilizzato anche come prima scelta di metodo di trasporto, quando precedentemente se ne incoraggiava l'uso esclusivamente in caso di fallimento di UDP. Conseguentemente ogni tipo di provider di servizi DNS deve poter accettare e servire richieste TCP, riutilizzando lo stesso stream con il quale il client ha effettuato la connessione.

Un server DNS dovrebbe lasciare al client il compito di aprire e chiudere ogni connessione, ma dovrebbe comunque implementare un sistema di timeout per evitare un potenziale spreco di risorse. L'implementazione di TCP su un server lo rende purtroppo fragile ad attacchi di Denial of Service, dato che un client malevolo potrebbe lasciare aperto un numero arbitrario di connessioni in corso. Non ci sono raccomandazioni particolari sul come difendersi da questo tipo di attacchi, se non generici suggerimenti come limitare il numero di connessioni o transazioni per indirizzo.

Il trasporto TCP prevede un overhead di connessione iniziale che ripetuto per ogni richiesta DNS potrebbe aggiungere una latenza rilevante. Sia client che server che scelgano di utilizzare TCP dovrebbero di conseguenza implementare il riuso di una stessa connessione persistente per tutta la durata della transazione, permettendo di inviare richieste e risposte multiple con un solo handshake TCP. È anche fortemente suggerita l'implementazione di pipelining: un client dovrebbe poter mandare sullo stesso stream più di una query senza dover attendere alcuna risposta, che il server dovrà gestire in concorrenza. Un server ricevente una pipeline non deve preoccuparsi di spedire ordinatamente le risposte, dovendo i client essere in grado di abbinare correttamente ogni risposta a una richiesta facendo uso dell'ID contenuto nell'header.

L'ultima parte fondamentale delle linee guida per TCP riguarda il formato del pacchetto. Un pacchetto DNS TCP possiede un campo iniziale relativo di 16 byte indicante la lunghezza totale del pacchetto DNS a livello applicazione. La ragione dietro a questo formato è che un segmento TCP potrebbe non riuscire a contenere un intero pacchetto o, al contrario, potrebbe contenere multipli pacchetti. Nella ricezione di un segmento DNS, il ricevente deve elaborare ogni pacchetto presente e assicurarsi di non chiudere la connessione (a parte in caso di timeout) prima di aver ricevuto un pacchetto nella sua interezza.

1.2.3 hashdns

Il proxy DNS di cui tratta l'elaborato deriva da un precedente tool sviluppato dal team di VirtualSquare, *hashdns*. [5, 26] HashDNS è un server DNS UDP non ricorsivo che si occupa di rispondere a query per indirizzi hash IPv6 basati su FQDN ed è il primo prototipo di DNS in grado di fornire questo servizio.

Il programma implementa un servizio DNS in grado di computare indirizzi hash e di risolvere query dirette o inverse di domini o indirizzi, come da specifiche, viste in

precedenza, per un risolutore idoneo. Purtroppo il funzionamento basilare richiede di impostare hashdns come server delegato per una famiglia di domini, quindi un primo setup non è immediato soprattutto a uso locale.

Essendo un risolutore indipendente e non un proxy non è necessario implementi le nuove feature di estensioni DNS, ma la mancanza di TCP in hashdns potrebbe limitare l'interazione con alcuni client. Un'altra falla del prototipo è la mancanza di asincronicità nella gestione delle richieste, che in situazioni di elevato carico potrebbe causare rallentamenti nella risoluzione di query.

1.3 IoTh, l'Internet of Threads

Una delle novità principali che si andranno a introdurre all'interno della progettazione e dello sviluppo dell'elaborato è l'utilizzo di stack di rete virtuali a livello di processo all'interno di un programma. La capacità di un'applicazione di gestire indipendentemente una propria interfaccia di rete apre le porte a nuove idee, superando le limitazioni imposte dal tipico scenario in cui si sarebbe altrimenti dipendenti dall'interfacciarsi con lo stack del kernel del sistema.

1.3.1 Processi come nodi di Internet

Il design originario e più comune del protocollo Internet prevede l'uso delle interfacce di rete fisiche degli host come endpoint delle connessioni. Questo significa che le entità indirizzabili sono a tutti gli effetti le macchine fisiche, anche se ciò che cercassimo di raggiungere fossero, come nella grande maggioranza dei casi, applicazioni o servizi contenuti al loro interno.

L'Internet of Threads (IoTh) è una proposta [4] di un modello in cui i processi possono essi stessi diventare entità indirizzabili di Internet. Come entità indirizzabile si intende a tutti gli effetti un endpoint della rete, con il proprio indirizzo IP e configurazione: mentre i limitati indirizzi di IPv4 ne avrebbero fortemente limitato l'usabilità (e probabilmente costretto all'uso di NAT), i pressoché illimitati indirizzi di IPv6 rendono realistica l'idea che ogni processo su una macchina possa essere singolarmente indirizzabile. Con questa logica è possibile costruire un'infrastruttura di rete più trasparente, in cui l'accessibilità a servizi e applicazioni è indipendente dalla macchina host su cui questi si trovano (ad esempio, una qualche applicazione server contenuta in un data center potrebbe mantenere sempre lo stesso indirizzo a prescindere dalla macchina su cui è eseguita) e in cui è possibile configurare routing di rete semplicemente con gerarchie di indirizzi IPv6, senza dover ricorrere a complesse configurazioni legate ai singoli host.

L'attuale proposta di implementazione di IoTh prevede l'uso di controller di rete virtuali per connettere i processi a switch (o indifferentemente hub) virtuali, che con permessi da amministratore possono essere a loro volta connessi alle interfacce di rete fisiche del dispositivo, divenendo a tutti gli effetti nodi di Internet. L'architettura di uno stack di rete così concepito prevede che ogni processo di IoTh abbia il proprio stack TCP-IP singolarmente configurabile, con cui possa connettersi a uno switch di rete virtuale senza nessun particolare privilegio. Con l'architettura di IoTh sarebbe ad esempio possibile che singoli processi sullo stesso host possano essere connessi a diversi

switch virtuali, in modo che ognuno di essi possa fare uso indipendente di una rete idonea alla loro attività: alcuni processi che scambiano informazioni critiche potrebbero essere connessi a un Virtual Private Network (VPN), mentre tutti gli altri potrebbero restare regolarmente connessi alla rete fisica.

Idealmente uno switch virtuale per IoTh dovrebbe essere in grado connettere interfacce e reti virtuali anche eterogenee e non direttamente compatibili tra loro. Il sistema in uso nello stato attuale di IoTh è il Virtual Distributed Ethernet (VDE) [7], che con l'interfaccia e utilità di sistema fornite da vdeplug4 [28] permette di interconnettere svariate tipologie di interfacce di rete virtuali, siano esse provenienti dallo stack del kernel (come un'interfaccia tuntap), da una macchina virtuale o da uno stack di rete virtuale di un processo. Questo è possibile grazie alla struttura modulare di VDE, che permette di implementare (tramite plug-in) e collegare tra loro moduli di rete differenti fornendo tool che implementano switch (o hub) virtuali. VDE fornisce in particolare un plug-in per collegarsi a un'interfaccia di rete tuntap del kernel, aprendo la possibilità di abilitare una qualunque altra rete virtuale compatibile all'accesso a Internet. Gli switch virtuali di VDE sono inoltre implementati a livello di utente, permettendo di attivarli, collegarli tra loro e collegarli a interfacce virtuali senza privilegi da amministratore.

L'indipendenza tra controller di rete del processo e switch permette costruzioni flessibili di uno stack di rete, che può essere interamente, parzialmente o non costruito all'interno del kernel. Lo sviluppo di uno stack di rete non è banale e per essere completo dovrebbe implementare interamente la API fornita da Berkeley sockets e un'interfaccia di configurazione del device di rete virtuale. Esempi correnti di stack di rete utilizzabili come librerie da processi, sviluppati dal team di VirtualSquare, sono lwIPv6 (basato su una fork dell'originale lwIP) [24] e picoxnet [25], implementati come stack TCP/IP indipendenti, oppure vdestack [23], implementato come stack TCP/IP appoggiandosi allo stack del kernel e facendo uso delle funzionalità dei namespace di rete. Mentre lwIPv6 e picoxnet sono al momento più sperimentali e incompleti, vdestack implementa interamente l'API di Berkeley sockets e permette una completa configurazione dell'interfaccia di rete tramite *netlink*. Gli stack virtuali sopracitati sono supportati da VDE, permettendoci di avere a disposizione un ambiente completo per un utilizzo basilare dell'idea di IoTh.

1.3.2 libioth

Lo sviluppo più recente dell'Internet of Threads è libioth [22], una libreria nata per implementare una API completa e flessibile per IoTh. Lo scopo della libreria è quello di unificare implementazioni eterogenee di librerie di stack di rete virtuali sotto un'unica API: libioth permette infatti di scegliere e configurare a runtime come plugin uno stack di rete virtuale a scelta, per poi usarlo con un'interfaccia di Berkeley sockets neutra indipendente dal tipo di stack.

La API di libioth mette a disposizione una funzione `ioth_newstack` per creare un nuovo stack di rete del tipo richiesto, con cui è poi possibile creare una *msocket* [8] sullo stesso con la chiamata `ioth_msocket`. Una volta creata la *msocket* e ottenuto un file descriptor è sufficiente utilizzare i wrapper della Berkeley sockets API, che resta invariata, con prefisso `ioth_` (`ioth_connect, ioth_send, ioth_recv`...) e si può ottenere un'applicazione che con naturalezza può fare uso di multipli stack di rete. La configurazione

di rete dello stack non necessita di un'implementazione particolare a parte ed è fornita tramite wrapping di *netlink* in un'altra API generica.

Il data link layer degli stack virtuali creati da libioth si appoggia a VDE. [7] Alla creazione di uno stack si specifica, oltre al plugin con cui crearlo, un *virtual network locator* (come in *vdeplug4* [28]), che identifica la rete VDE alla quale esso verrà connesso, e in aggiunta, facoltativamente, una serie di nominativi di interfacce di rete da creare. In questo modo anche la gestione della rete virtuale su cui si trova lo stack creato è rilegata a una API unica che prescinde dal tipo di *virtual network locator* scelto.

La flessibilità offerta da libioth permette di costruire programmi che possano funzionare sia come di norma sullo stack del kernel, specificando come stack da utilizzare appunto "kernel", sia su uno stack di rete virtuale, sia in ibrido. Questa flessibilità ha due importanti implicazioni. Prima di tutto si può scegliere di implementare un qualunque programma con libioth in modo che sia automaticamente adatto a un uso regolare sullo stack del kernel e allo stesso tempo compatibile per un uso basato su IoTh. In secondo luogo un programma può utilizzare stack di rete ibridi e conseguentemente esistere su due data plane diversi: quello fisico di una rete di calcolatori e quello dell'Internet of Threads.

1.3.3 One Time IP Address

Una delle idee portate alla luce dall'utilizzo di IPv6 e dall'architettura dell'Internet of Threads è OTIP (One Time IP Address) [6], con cui si intende la capacità di un server di modificare periodicamente il proprio indirizzo IP per proteggersi da attacchi. Un server OTIP computa periodicamente e deterministicamente il suo indirizzo sulla base di informazioni possedute dagli utenti abilitati al suo uso, tipicamente una password e l'orario: ciclando periodicamente in uno spazio di indirizzamento IPv6 di 64 bit (che come visto precedentemente è il minimo suggerito anche per una rete IPv6 casalinga [19]) si protegge da potenziali attaccanti e rende inutili ricerche di indirizzi sullo spazio dei possibili 2^{64}, data la volatilità di questi. Lo scopo di OTIP è aggiungere un ulteriore layer di sicurezza, che impedisca ai potenziali attaccanti di poter semplicemente bersagliare server protetti con indirizzi noti con attacchi di forza bruta o sfruttando altri tipi di vulnerabilità. Infatti, anche se un attaccante riuscisse a trovare l'indirizzo del server protetto da OTIP, l'indirizzo sarebbe valido solo per un periodo di tempo limitato prima di venire nuovamente computato.

L'attuale proposta implementativa di OTIP prevede che l'indirizzo IPv6 del server venga computato sulla base del suo FQDN, una password e l'orario corrente. Basare la computazione su un orario richiede che i Real Time Clock (RTC) di server e client siano il più possibile sincronizzati, poiché un client non correttamente sincronizzato con l'orario del server non sarebbe in grado di computare l'indirizzo di accesso corretto. È scontato che l'assunzione di perfetta sincronizzazione tra server e potenziali multipli client sia assurda: sia l'imperfetta sincronizzazione dei RTC, sia la latenza introdotta dalla rete, è necessario un grado di tolleranza nell'accesso al server OTIP. Data una sequenza di indirizzi computati dal server $a_1, \cdots a_n$, deve esserci un arco temporale in cui due indirizzi consecutivi a_{n-1}, a_n siano validi contemporaneamente per sopperire alle inevitabili imprecisioni dei sistemi. Sia questo arco temporale Δ_t il valore di massima desincronizzazione tra l'orario di un client e del server, t_c il valore dell'orario di un

client a tempo t e t_s quello del server, allora: $\forall c : |t_c - t_s| < \Delta_t$. Una sovrapposizione di indirizzi accessibili di questo genere può quindi facilmente risolvere gli eventuali problemi di sincronizzazione che possono essere creati da variabili fuori dal controllo del protocollo di OTIP. Attualmente la generazione di indirizzi OTIP può essere vista come un caso particolare di quella degli indirizzi hash basati su FQDN [5], potendo computarli con la funzione di hashing vista in precedenza aggiungendo lo XOR di un digest della password e di un digest di $TIME\%TIMESLICE$, dove TIME sono i secondi dalla Unix epoch e TIMESLICE è la durata in secondi dell'indirizzo temporaneo.

Sarebbe possibile implementare un server OTIP su un comune host facendo uso di un demone che, aggiungendo e rimuovendo indirizzi da un'interfaccia di rete fisica o virtuale, riproduca le specifiche descritte; questo demone richiederebbe tuttavia diritti di amministrazione di rete. In questo caso il processo server dovrebbe solamente selezionare l'indirizzo corretto tramite uso della system call `bind`. In realtà l'idea su cui si vuole basare OTIP allo stato attuale è quella di utilizzare le potenzialità di IoTh: se un processo implementa direttamente al suo interno uno stack di rete, allora può gestire senza diritti speciali e in completa autonomia le specifiche del protocollo di OTIP. Un'implementazione di questo tipo permette anche la coesistenza di più server OTIP sullo stesso host, ciascuno con la propria configurazione di durata di un indirizzo, tempo di sovrapposizione di indirizzi, password di computazione... che non sarebbe altrimenti possibile con il metodo precedente.

Nel concreto, è possibile costruire un'istanza di OTIP facendo uso di un risolutore DNS specializzato, che effettui la computazione degli indirizzi corretti per i client, e di un server abilitato alla computazione di indirizzi temporanei. Realisticamente, richiedere che un generico server sia abilitato nativamente all'uso dell'algoritmo di OTIP per poterne usufruire è troppo limitante. È possibile avere un reverse proxy abilitante, che si occupi della gestione delle connessioni dall'esterno facendo forwarding verso dei server interni che vogliano essere protetti da OTIP: in questo modo un qualunque server potrebbe usufruire del sistema senza dover essere modificato. Come gestire però le connessioni in corso quando avviene una ricomputazione dell'indirizzo IP? Nel caso di un protocollo connection-oriented come TCP è sufficiente mantenere aperto uno stack di rete finché vi restino connessioni attive, chiudendo solamente il socket in ascolto e quindi impedendo di accettare nuovi client su uno stack scaduto. Purtroppo nel caso del connection-less UDP la questione non è altrettanto semplice, dato che non è possibile mantenere aperto un flusso di dati. In questa istanza sarebbero necessari o dei client OTIP-aware o dei proxy tunnel UDP che si occupino di indirizzare i pacchetti di un client all'indirizzo corretto.

Si prova in seguito a illustrare un esempio di funzionamento di server TCP (figura 1.2) basato sui concetti appena esposti. Come infrastruttura abbiamo un server con un FQDN aperto in una rete interna chiusa (ad esempio, la porta su cui il server accetta connessioni è chiusa per l'esterno), un reverse proxy OTIP in grado di connettersi al server e una serie di client esterni con accesso a un DNS privato (ad esempio, un proxy locale) in grado di risolvere domini OTIP. Il reverse proxy computa un nuovo indirizzo allo scadere di ogni timeslice e apre il nuovo stack corrispondente su cui inizia ad accettare nuove connessioni. Quando un client si connette, la connessione viene reindirizzata verso il server interno che resta inconsapevole del layer OTIP in uso. Il proxy mantiene aperto uno stack finché almeno un client vi è ancora connesso, ma allo scadere dell'indirizzo relativo chiude il socket di accettazione in modo da rifiutare nuove connessioni. Nel mo-

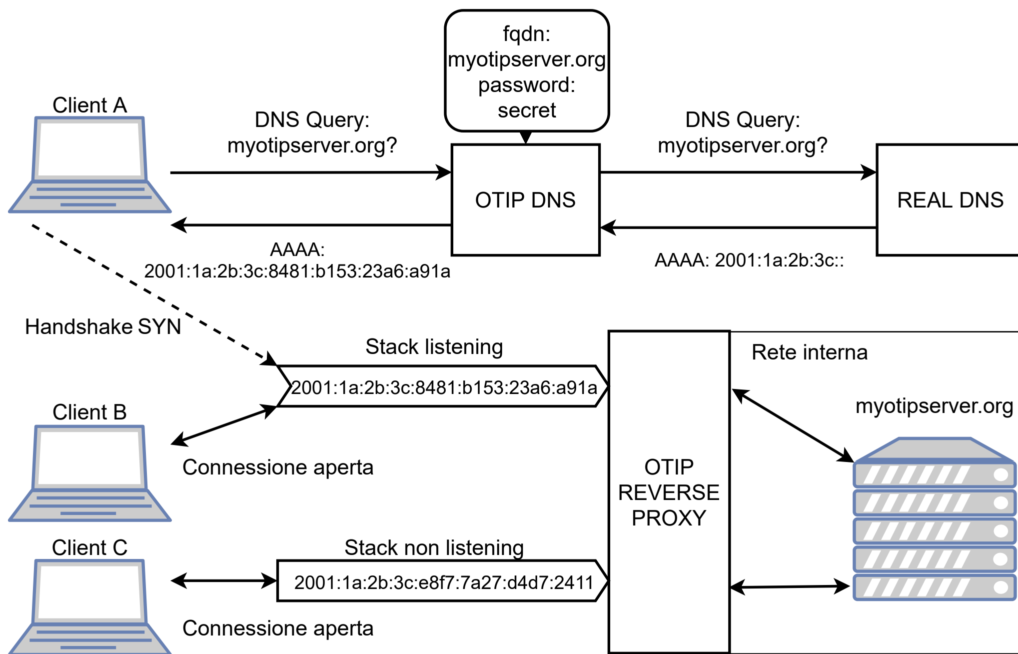


Figura 1.2: Esempio dinamica OTIP

mento un nuovo client voglia connettersi, manda la richiesta a un DNS specializzato che, avendo a disposizione FQDN, password e timeslice, può risolvere l'indirizzo temporaneo direttamente, se in possesso dell'indirizzo base relativo al dominio, o inoltrando prima la richiesta a un master DNS. Per evitare caching degli indirizzi temporanei, dovrebbe essere premura del DNS OTIP di specificare un Time To Live (TTL) di un secondo per il record AAAA di risposta.

È chiaro come OTIP possa funzionare come layer di sicurezza aggiuntivo per la protezione di servizi privati, privando fin dal principio gli attaccanti della chance di bersagliarne potenziali vulnerabilità. Purtroppo, per quanto si sia osservato che l'attuale stato dell'arte ne consenta una reale implementazione, al momento gli strumenti necessari precedentemente descritti sono solo proof-of-concept dimostrativi.

Capitolo 2

Sviluppo di VdeDNS

2.1 Analisi progettuale

2.1.1 Finalità

L'implementazione delle idee precedentemente introdotte di indirizzi hash IPv6 basati su FQDN e di indirizzi IPv6 temporanei non presenta implementazioni complete all'attuale stato dell'arte. L'infrastruttura di una rete a indirizzi hash con host autoconfiguranti richiede un risolutore specializzato, in grado di rispondere alle query di dominio pertinenti, e un sistema di autoassegnamento dell'indirizzo in base a un nome; un layer di sicurezza basato su indirizzi temporanei richiede un generatore di indirizzi (e un eventuale tunnel per l'utilizzo di UDP) client side e un server dedicato o un reverse proxy abilitante server side. Possiamo colmare immediatamente gran parte delle mancanze attraverso DNS: l'infrastruttura a indirizzi hash può appoggiarsi a un server DNS in grado di computare indirizzi basati su FQDN e di memorizzare in cache domini per risoluzioni inverse, mentre un client che desideri connettersi a un server abilitato a indirizzi temporanei può semplicemente affidarsi a un server (o proxy locale) DNS che esegua la computazione dell'indirizzo.

Lo scopo di questa sezione è presentare un tentativo di riempire uno dei principali tasselli necessari alla completa realizzazione delle idee presentate finora. VdeDNS è un proxy DNS sviluppato allo scopo di implementare e rendere facilmente accessibili le feature degli *IPv6 hash-based address* (abbreviati *HASH*) e dei *one time ip address* (abbreviati *OTIP*), con una quantità minimale di overhead e configurazione. La principale funzionalità del proxy vuole essere risolvere le richieste verso domini preconfigurati come *HASH* e/o *OTIP* come conforme alla loro specifica, utilizzando indirizzi base interni se configurati, altrimenti ottenendo gli indirizzi base relativi al dominio richiesto inoltrando la query a un master DNS. In aggiunta, VdeDNS vuole essere pensato per poter servire nativamente processi dell'Internet of Threads e vuole quindi essere anch'esso conseguentemente abilitato al funzionamento su stack di rete virtuali.

Come proxy DNS, il programma deve farsi carico di tutte le richieste di risoluzione ricevute a prescindere dal fatto che queste siano o meno di sua competenza. Questo comporta, per un corretto funzionamento, che un utente utilizzatore non debba notare alcuna differenza nel meccanismo di risoluzione di query comuni rispetto a quando non fa utilizzo del proxy: è dunque necessario che il forwarding di richieste non *OTIP*/*HASH*

avvenga nel modo più diretto e trasparente possibile. Congiuntamente, ci si aspetta che anche le query speciali di competenza del DNS siano gestite in linea con il protocollo, entro le loro funzionalità. Le query che VdeDNS può dover gestire sono dirette o inverse IPv6, quindi record (potenzialmente multipli) AAAA, come nei casi base di una richiesta di un indirizzo *HASH* o *OTIP*, o PTR, come nel caso più specifico di una richiesta inversa di un dominio *HASH*. Per massima compatibilità con i protocolli di rete attuali, il proxy DNS dovrebbe essere inoltre sviluppato in concordanza con le linee guida indicate nell'RFC5625. [1] Queste linee guida forniscono indicazioni generali sul come prevenire incompatibilità nei casi di forwarding diretto delle richieste, in modo che nel momento in cui il proxy si ritrovi a risolvere richieste non di sua specifica competenza (quindi tendenzialmente nella maggioranza dei casi) queste non vengano ostacolate.

Un processo dell'Internet of Threads si trova tipicamente su un proprio stack di rete contenuto in una rete virtuale. Per dargli accesso a un comune proxy DNS, in ascolto sullo stack di rete del kernel, la rete virtuale dovrebbe essere collegata direttamente a quella fisica (collegamento che richiede necessariamente permessi da amministratore) o alternativamente ospitare il proxy su una VM a essa collegata. VdeDNS, essendo specializzato nel servire processi IoTh, dovrebbe poter essere eseguito a sua volta come un processo IoTh con il proprio stack di rete, permettendo un setup non impegnativo anche su una rete virtuale chiusa, se fosse necessario. Utilizzando stack di rete virtuali è inoltre possibile che un programma sia in grado di estendersi su reti diverse: nel caso di un proxy DNS si potrebbero ad esempio avere i file descriptor in ascolto su uno stack separato dai file descriptor usati per il forwarding di richieste, creando un potenziale layer aggiuntivo di sicurezza e flessibilità.

2.1.2 Scelte implementative

Linguaggio C

Il linguaggio scelto per l'implementazione di VdeDNS è il C. Le ragione principale dietro a questa scelta è molto naturale: il supporto usato per le feature di virtualizzazione degli stack di rete è *libioth*, che è una libreria C. L'utilizzo di un linguaggio di basso livello per l'implementazione di un sistema di rete permette in ogni caso anche una libertà molto estesa riguardo la gestione della connettività, la virtualizzazione, e il parsing di pacchetti. Lavorando a basso livello sulle API di rete la gestione delle trasmissioni di dati avviene in maniera trasparente, permettendo accorgimenti sull'apertura di socket, apertura di connessioni e passaggio di dati.

Con la API offerta da *libioth* la gestione degli stack di rete virtuali è pressoché invariata rispetto alla norma, avendo a disposizione tutte le funzioni standard dell'API C di Berkeley sockets indipendentemente che si stia usando lo stack di rete del kernel o uno stack di rete virtuale. Per questa ragione VdeDNS utilizza esclusivamente (con unica eccezione di socket per inter-process communication) la API di rete di *libioth*, che consente di eseguire il proxy come un regolare processo sullo stack del kernel, interamente virtualizzato o parzialmente virtualizzato.

Per il parsing di pacchetti si è scelta di utilizzare la libreria in sviluppo *iothdns*, che fornisce un'interfaccia basata su file stream per fare parsing e ricomposizione di pacchetti a basso livello.

Oltre ai vantaggi di lavorare a basso livello, un eseguibile compilato dal C è generalmente molto veloce e a basso consumo di memoria: la velocità è un requisito per l'elaborazione interna di pacchetti DNS perché il servizio sia usufruibile in casi d'uso con più esigenze di un semplice utilizzo personale, mentre il basso consumo può permettere un utilizzo poco invasivo del proxy anche su macchine poco performanti. Non solo, ma la compilazione dal C permette compatibilità con un ampio numero di differenti sistemi e architetture.

Feature DNS

Le scelte implementative riguardanti le feature del proxy DNS sono state principalmente derivate dall'RFC5626 [1], come accennato in precedenza. Per prima cosa, VdeDNS esegue forwarding diretto di qualunque pacchetto di query non sia di sua competenza. Un pacchetto è considerato di competenza del proxy quando il dominio richiesto fa matching con una delle regole all'interno della configurazione, quindi nel caso sia un dominio configurato come *HASH* o *OTIP*, oppure se sia un dominio con dei record associati internamente. Tutti gli altri casi sono considerati pacchetti generici, a unica condizione che presentino ovviamente una flag da query, e come da indicazione vengono mandati a un master DNS senza alcuna modifica alle loro flag o record presenti. Non solo flag e record non vengono modificati, essendo poco realistico il poter implementare e prevedere tutti i possibili protocolli DNS, il proxy non rifiuta pacchetti con flag o record sconosciuti. L'unica modifica effettuata a ogni pacchetto, generico o meno, è il cambio di ID prima del forwarding: questa modifica è perfettamente ammissibile dallo standard a patto che, per ragioni di sicurezza, questi ID siano generati con un livello di entropia sufficientemente elevato. Gli ID sono dunque generati utilizzando le funzioni `random()` e `getrandom()`, che generano numeri casuali non linearmente e con un elevato livello di entropia. Come per le richieste, anche le risposte generiche vengono rimandate al client senza alcuna variazione nei loro contenuti. Il forwarding viene eseguito in ordine a round robin su una lista di server DNS forniti nella configurazione: se uno di questi non dovesse rispondere entro un timeout fissato, viene interrogato il successivo.

L'implementazione di TCP in aggiunta al già standard UDP è un requisito sui proxy DNS. La ragione principale dietro a questa richiesta è, oltre alla recente crescita della diffusione di TCP come protocollo di trasporto di pacchetti DNS, che TCP è un meccanismo di fallback in caso che un pacchetto UDP di risposta ecceda la dimensione limite. In questa occasione, il protocollo DNS prevede che il pacchetto di risposta venga troncato e che vi venga acceso un bit detto di troncamento. Quando un client riceve un pacchetto troncato, rimanda la stessa query cambiando il metodo di trasporto a TCP, che consente pacchetti fino a 65536 byte. È quindi importante che per prevenire fallimenti in questi casi (che con la diffusione di IPv6 e nuove funzionalità dei DNS sono sempre più comuni) un proxy DNS sia in grado quantomeno di eseguire forwarding attraverso TCP. Per questa ragione, VdeDNS opera su due thread separati due servizi completamente funzionali di UDP e TCP. In aggiunta, implementa anche il meccanismo di troncamento per UDP appena descritto nel caso che i pacchetti di risposta da esso generati superino in dimensione la soglia imposta. VdeDNS è pensato per elaborare richieste relative a IPv6, quindi è naturale prevedere che record AAAA contenenti multipli indirizzi possano eccedere i limiti di UDP.

Con l'introduzione dei meccanismi di estensione per il protocollo DNS [3], sono richiesti alcuni cambiamenti agli standard iniziali per poter mantenere la compatibilità con nuove feature potenziali o già in circolazione. Nella prima stesura degli standard del protocollo, la dimensione massima di un pacchetto UDP è definita come 512 byte. Purtroppo con l'introduzione delle estensioni DNS questo limite è spesso non più sufficiente e sarebbe richiesto che un proxy DNS sia in grado di supportare pacchetti fino ad almeno 4096 byte senza che questi vengano troncati. A questo riguardo, si è scelto di mantenere il limite di default di VdeDNS a 512 byte, aggiungendo però l'opzione di estenderlo al lancio del programma nel caso sia necessario. Le estensioni introducono anche il nuovo tipo di RR OPT, che può trovarsi anche all'interno di una query. VdeDNS supporta il forwarding diretto di questi pacchetti come da istruzioni, ma non si preoccupa di seguirne le indicazioni: non vengono infatti generate risposte ad hoc per alcun tipo di estensione, a meno che non provengano da un master DNS.

VdeDNS è un prototipo costruito allo scopo di sperimentare le nuove feature di risoluzione tramite hashing di indirizzi e di virtualizzazione: per questa ragione l'implementazione di alcune funzionalità non strettamente necessarie aggiuntive è stata omessa. Per quanto potrebbe in futuro essere una feature utile, il proxy attualmente non fa utilizzo di alcuna forma di caching (se non per la risoluzione inversa di indirizzi *HASH*). Anche la risoluzione ricorsiva è stata omessa, principalmente perché essendo il programma un proxy DNS è risultato più ragionevole riuscire a mantenere trasparenza e compatibilità facendo forwarding di richieste a un DNS reale. Riguardo a feature di sicurezza più recenti come DNSSEC e DNS over TLS, si è scelto di non implementarle direttamente. Il proxy è in grado di fare forwarding di pacchetti DNSSEC per query non di sua competenza (purché il buffer impostato per UDP sia sufficientemente ampio), ma non è in grado di validare risposte costruite in autonomia. DNS over TLS è invece incompatibile con il proxy, poiché richiederebbe ricezione e parsing su porta 853 di pacchetti criptati e conseguentemente un forwarding diretto non sarebbe possibile. Non è da escludere che quest'ultima feature possa essere implementata in futuro utilizzando l'infrastruttura TCP già costruita.

Design Asincrono

Uno dei requisiti principali di un server DNS è l'asincronia delle operazioni. Le specifiche iniziali [16] e evoluzioni successive [10] del protocollo prevedono che un server processi parallelamente e in modo non bloccante ogni richiesta. In particolare, nella costruzione di VdeDNS si è scelto di utilizzare un design basato su una gestione a eventi. Una valida alternativa sarebbe stata basarsi sull'uso di thread, ma si è preferito utilizzare una soluzione più leggera dal punto di vista del consumo di memoria, anche se a costo di complicare la struttura del codice.

L'implementazione del lato UDP a eventi è triviale: si devono solo tenere sotto controllo un singolo socket per la ricezione di richieste e un singolo socket per la ricezione di risposte da parte dei master DNS. Per questa ragione UDP permette di usare senza alcuna perdita di prestazioni la chiamata `poll`, che risulta in una gestione veloce e con poco overhead degli eventi.

Purtroppo una gestione a eventi su un protocollo non connection-less come TCP è più complessa da scalare: il file descriptor di accettazione genera un nuovo file descriptor

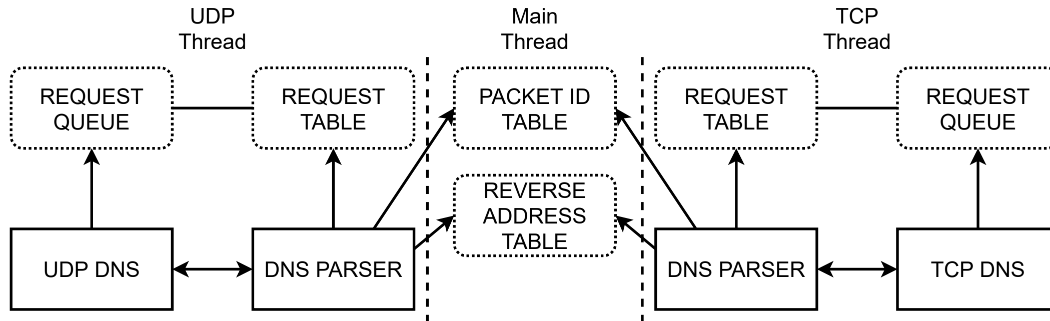


Figura 2.1: Struttura generale di VdeDNS

a ogni connessione e ogni master DNS richiede a sua volta il proprio file descriptor per ricevere e rispondere a richieste di forwarding. La chiamata `poll` sarebbe stata ovviamente utilizzabile anche in questa istanza, ma non senza alcuni compromessi. Per prima cosa `poll` opera su un insieme statico di file descriptor definiti in un vettore, conseguentemente sarebbe necessario allocare un vettore sufficientemente grande per tutte le possibili connessioni o alternativamente riallocarlo dinamicamente ripetendo poi ogni volta la chiamata di funzione. Inoltre la chiamata non ha modo di restituire informazioni al di fuori del file descriptor risvegliato e della natura dell'evento associato, che sono invece indispensabili in una gestione asincrona di diverse connessioni TCP, che richiederebbero quindi strutture dati separate.

Questi problemi sono risolvibili implementando il design a eventi di TCP con la chiamata `epoll`, che gestisce l'allocazione e restituzione di strutture dati anche arbitrarie a livello kernel. La `epoll` risulta essere più scalabile e performante [11] della precedente `poll` anche per come avviene il monitoraggio dei file descriptor: mentre `select` e `poll` devono richiedere al kernel a ogni chiamata di monitorare l'insieme di tutti i file descriptor, non venendo questi memorizzati (sono quindi sistemi di monitoraggio stateless), la `epoll` permette di assegnare dinamicamente un file descriptor al kernel (o di modificarne o rimuovere uno già monitorato) chiamando `epoll_ctl` e di farsi restituire esclusivamente informazioni riguardanti a file descriptor su cui sono avvenuti eventi chiamando `epoll_wait`.

2.2 Implementazione

2.2.1 Struttura di VdeDNS

VdeDNS (figura 2.1) è eseguito su tre thread principali: il main thread, il thread dedicato a UDP e il thread dedicato a TCP. Sul main thread risiedono alcune strutture dati e informazioni in comune per l'elaborazione di pacchetti dell'intero proxy, sui thread dedicati ai protocolli risiedono strutture dati speculari a uso locale del thread. Mentre l'accesso alle strutture dati in comune è regolato da mutua esclusione, l'accesso alle strutture locali dei thread UDP e TCP è costruito in modo da non avere potenziali accessi multipli.

Ad eccezione dell'header indicante la dimensione di un pacchetto su TCP, i pacchetti DNS di UDP e TCP sono strutturalmente identici ed è conseguentemente possibile ela-

borarli nello stesso modo. Tuttavia, i protocolli di connettività sono invece differenti dal punto di vista implementativo e difficili da fare coesistere. Per questa ragione VdeDNS è strutturalmente pensato per separare completamente l'elaborazione e formazione di pacchetti DNS dagli aspetti di ricezione e invio degli stessi. Nel dettaglio, le funzioni di parsing ricevono in input un pacchetto DNS generico, con eventuale header TCP rimosso, e le funzioni di forwarding e di invio risposta da utilizzare al termine dell'esecuzione. In questo modo è possibile modificare liberamente l'implementazione dei protocolli di connettività senza che sia necessario modificare la struttura delle funzioni di parsing.

2.2.2 Strutture dati principali

Un servizio DNS deve tipicamente porre enfasi sulla velocità di elaborazione delle richieste ed essere il più possibile fault-tolerant. Una parte essenziale per soddisfare questi requisiti è una corretta scelta dell'implementazione delle strutture dati del programma. VdeDNS fa uso estensivo di strutture tabulari come insiemi e tabelle hash, per favorire un accesso immediato ai dati anche in presenza di eventuali carichi elevati di richieste. Le principali strutture dati comuni del proxy sono:

ID Table Tabella di ID correntemente in uso per il forwarding

Reverse Address Table Lista delle coppie (indirizzo, FQDN) per risoluzione inversa di indirizzi hash

Request Table Tabella delle richieste DNS in attesa di una risposta dal master DNS

Request Queue Code delle richieste DNS in attesa di una risposta dal master DNS

Come si può notare la Request Table e la Request Queue contengono gli stessi dati: sono infatti in realtà un'unica struttura collegata, accessibile a seconda della situazione come tabella hash o come coda.

ID Table

Lo scopo della ID Table è prevenire conflitti di ID tra le richieste in corso verso i master DNS. Se questo problema fosse ignorato, due pacchetti potrebbero essere riinviati al mittente errato, venendo questi identificati in base al loro ID (il problema è noto e viene citato nell'RFC5625 [1]). Essendo la sezione ID dell'header di un pacchetto DNS lunga 16 byte [16], la quantità di possibili combinazioni è ridotta a 65536: è quindi ragionevole pensare di implementare una struttura di tipo insieme per verificare quali di questi pacchetti sia in utilizzo durante l'esecuzione.

L'interfaccia della ID table fornisce semplicemente due funzioni: `get_unique_id`, che prova a restituire un ID unico al chiamante e ne aggiorna l'uso, e `free_id` che libera un ID in uso una volta che il pacchetto è ritornato dal master DNS.

Listing 2.1 : get_unique_id

```
#define MAX_RETRY 8
uint16_t get_unique_id(){
  int i;
```

```
  uint16_t id;
  pthread_mutex_lock(&idlock);
  for(i = 0; i < MAX_RETRY; i++){
      id = random();
      if(id_table[id] == 0) {
          id_table[id]++;
          break;
      }
  }
  if(i >= MAX_RETRY) {
      printlog(LOG_ERROR, "Failed␣to␣generate␣unique␣ID.\n");
      id_table[id]++;
  }
  pthread_mutex_unlock(&idlock);
  return id;
}
```

La funzione `get_unique_id` lavora ottimisticamente e tenta di generare un ID casualmente, controllando poi se sia o meno in uso nell'insieme. Nel caso l'ID generato sia già in uso, la funzione ritenta un numero massimo di volte per poi arrendersi e reciclare un elemento già in uso dall'insieme. Il limite imposto al numero di tentativi è presente per proteggersi da casi di estrema congestione: se approssimativamente tutti gli ID fossero impegnati, la ricerca ottimistica potrebbe causare un bottleneck superiore a una ricerca lineare di un ID libero. È ovviamente un caso estremamente raro che tutti o quasi i 2^{16} ID siano occupati, ma non del tutto impossibile nell'istanza di un elevato numero di richieste unite a diversi fallimenti o ritardi sequenziali dei master DNS. Un caso comunque raro ma più realistico è che approssimativamente la metà degli ID siano impegnati. In questa situazione, a ogni tentativo al 50% delle possibilità verrà ottenuto un ID non valido, ma con anche solo un basso numero di tentativi, come ad esempio 8, le chance di ottenere un ID non valido diventano $\frac{1}{2}^8 = \frac{1}{256} = 0.0039$, approssimativamente 0.4%. In ogni caso un fallimento della ID table non è detto comporti un conflitto, per questa ragione si è ritenuto il metodo ottimistico superiore a una ricerca lineare.

Reverse Address Table

La Reverse Address Table in uso da VdeDNS è una versione aggiornata dello stesso codice presente su HashDNS, ma reso thread safe tramite wrapping delle funzioni originali. In aggiunta sono state spostate nell'interfaccia alcune funzioni di parsing precedentemente dichiarate altrove, leggermente adattate al formato della libreria *iothdns* attualmente in uso.

La struttura dati principale è una semplice lista concatenata di strutture `revaddr`, che contengono al loro interno un indirizzo IPv6, un FQDN e una scadenza in secondi. Le funzioni `ra_add`, `ra_search` e `ra_clean` rispettivamente aggiungono una nuova coppia FQDN,indirizzo, cercano un indirizzo tramite un FQDN e ripuliscono la lista dalle coppie con tempo scaduto. La Reverse Address Table è unica all'interno del programma, conseguentemente queste funzioni utilizzano lock di mutua esclusione per prevenire conflitti tra i thread.

Come suggerito dalle specifiche date per gli indirizzi *HASH* [5], la Reverse Address Table permette di scegliere una politica di filtraggio di indirizzi tra "ALWAYS", "SAME" e "NET": nel primo caso ogni indirizzo di tipo *HASH* verrà aggiunto alla struttura, negli altri due casi verranno aggiunti esclusivamente se il client richiedente sia lo stesso indirizzo risolto o se appartenga alla stessa rete /64, rispettivamente.

Request Table e Request Queue

La Request Table e la Request Queue sono un'unica struttura dati che inserisce contemporaneamente ogni elemento in una tabella hash e in una coda. Questo tipo di struttura è specificamente pensato per tipi di risorse che possono essere accedute univocamente tramite un qualche tipo di identificativo e che allo stesso tempo devono restare nella struttura dati per al più un dato limite di tempo. L'esempio qui trattato riguarda le richieste DNS in forwarding, ma la struttura viene riutilizzata dall'implementazione TCP per controllare il timeout dei file descriptor. La Request Table e Queue sono infatti semplicemente dei wrapper costruiti intorno a una più generica struttura dati `hashq`.

Listing 2.2 : struct hashq

```c
struct hashq {
  struct hashq *qnext;
  struct hashq *qprev;

  struct hashq *hnext;
  struct hashq *hprev;

  long expire;
  void* data;
};
```

Ogni elemento della struttura ha un collegamento bidirezionale alla lista di trabocco della sua posizione rispettiva nella tabella hash, un collegamento bidirezionale alla coda, una scadenza e un puntatore generico ai dati che contiene. L'accesso a un elemento attraverso il suo identificativo nella tabella hash avviene approssimativamente in $O(1)$, in particolare proprio nel caso della Request Table. La tabella hash delle richieste è in realtà implementata come una struttura insieme, con 2^{16} elementi nel vettore (uno per ogni possibile ID DNS), di conseguenza, ad esclusione di fallimenti nella generazione di ID di pacchetti, il tempo di accesso è esattamente costante. Anche la rimozione di un singolo elemento scaduto avviene in tempo costante: si presuppone che per ogni elemento $i, j, j > i$, j inserito successivamente nella coda abbia un valore di expire più alto di i inserito precedentemente. Per questa ragione, se estraendo il primo elemento dalla coda questo non è scaduto, allora nemmeno tutti gli elementi successivi saranno scaduti. Questo permette quindi una ricerca efficiente degli elementi da rimuovere dalla struttura, rimuovendo uno a uno dalla testa della coda tutti quelli scaduti fino al primo ancora valido. Facendo uso dei collegamenti bidirezionali, la rimozione di un elemento tramite identificativo ha costo ugualmente costante: è sufficiente accedervi tramite tabella hash e ricollegare tra loro l'elemento precedente e quello successivo a esso nella coda e nella lista di trabocco.

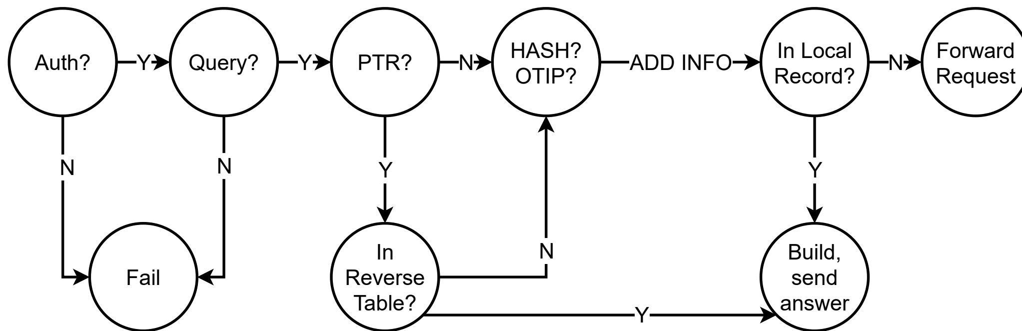


Figura 2.2: Procedura di Query Parsing

2.2.3 DNS Parser

Il parsing di un pacchetto DNS ha due differenti punti di ingresso: il primo è `parse_req` , per le query ricevute dai client, il secondo è `parse_ans`, per le risposte ricevute dai master DNS. Entrambe le funzioni ricevono in input il buffer di un singolo pacchetto, la sua lunghezza e le funzioni da chiamare nel momento in cui si debba mandare una risposta a un client o fare forwarding a un master DNS. La ragione per cui vengono passate in input le funzioni di trasmissione dati è, come visto in precedenza, per separare il livello di elaborazione del pacchetto, che è unico a prescindere dal protocollo di rete utilizzato, dal livello di gestione della connettività, che differisce tra UDP e TCP.

La funzione di parsing di richieste (figura 2.2) si assicura in primo luogo che il pacchetto sia da una fonte autorizzata, nel caso VdeDNS sia stato eseguito con l'opzione di verifica di autorizzazione, e che il pacchetto sia effettivamente di tipo query. Successivamente controlla se il dominio richiesto sia di tipo PTR e nel caso ne verifica l'appartenenza alla Reverse Address Table; in caso positivo costruisce il pacchetto di risposta, lo invia e ritorna. Nel caso sia un altro tipo di richiesta, viene verificato se il dominio appartenga invece a una (o entrambe) le categorie *HASH* e *OTIP*, aggiungendo queste informazioni a una struttura dati relativa alla richiesta. In particolare, gli indirizzi di tipo *HASH* hanno tipicamente un suffisso aggiuntivo rispetto al FQDN di base con cui possono fare match, conseguentemente un eventuale pacchetto di forwarding deve contenere la query verso il dominio base e non quello intero, che non otterrebbe risultati. È quindi necessario salvare il FQDN intero della richiesta, per poter inviare la risposta al mittente con il dominio corretto. Se il dominio richiesto appartiene alla lista di domini gestiti localmente da VdeDNS, allora viene costruito un pacchetto di risposta, eseguendo le eventuali trasformazioni per *HASH* e *OTIP*, e rispedito direttamente al client richiedente. Nel caso contrario, se il proxy è eseguito in modo da consentire il forwarding, viene costruito e inviato un pacchetto di richiesta per un master DNS e i dati relativi alla richiesta vengono inseriti nella Request Queue/Table. Il pacchetto di richiesta è costruito assicurandosi di trasferire eventuali RR aggiuntivi e modificando l'ID con uno generato tramite la ID Table.

La funzione di parsing di risposte va innanzitutto alla ricerca di informazioni relative alla richiesta in sospeso confrontando ID e query name del pacchetto di risposta con il contenuto della Request Table e nel caso non trovasse un match lo ignora. Questo caso è per esempio comune nella situazione in cui un master DNS tardi nella risposta e VdeDNS abbia ritenuto la richiesta di forwarding già scaduta. Se trova informazioni

relative alla richiesta in sospeso e questa non ha informazioni relative a *HASH* o *OTIP*, il pacchetto viene spedito direttamente al client reinserendo solamente l'ID della richiesta iniziale. Se il pacchetto è invece di competenza di VdeDNS, viene costruita una risposta eseguendo la trasformazione degli indirizzi IPv6 ricevuti in risposta secondo le regole di *HASH* e *OTIP*, nel caso di *HASH* viene reinserito il query name originale e come nel caso precedente viene rimesso l'ID atteso dal client.

Listing 2.3 : solve_hashing

```
static void solve_hashing(struct pktinfo* pinfo){
  if(pinfo->h->qtype == IOTHDNS_TYPE_AAAA){
    int i;
    for(i=0; i < pinfo->addr_n; i++){
      if(pinfo->type & TYPE_OTIP){
        iothaddr_hash((void*)&pinfo->baseaddr[i],
            pinfo->h->qname, pinfo->opt,
            iothaddr_otiptime(pinfo->otip_time ?
            pinfo->otip_time : DEF_OTIP_PERIOD, 0));
      }
      else if(pinfo->type & TYPE_HASH){
        char buf[IOTHDNS_MAXNAME];
        if(get_subdom(buf, pinfo->origdom, pinfo->h->qname) > 0){
            iothaddr_hash((void*)&pinfo->baseaddr[i],
                pinfo->origdom, NULL, 0);
        }
      }
      pinfo->rr = malloc(sizeof(struct iothdns_rr));
      *pinfo->rr = (struct iothdns_rr){.name=pinfo->h->qname,
        .type=IOTHDNS_TYPE_AAAA, .class=IOTHDNS_CLASS_IN,
        .ttl=(pinfo->type & TYPE_OTIP) ? 1 : TTL};
    }
  }
}
```

La funzione `solve_hashing` trasforma gli indirizzi di un pacchetto marcato *HASH*/*OTIP* utilizzando le funzioni di hashing fornite dalla libreria *iothaddr*. Nel caso di un tipo *HASH* si controlla se il dominio sia un dominio base o con un suffisso. Un dominio base viene lasciato invariato, in modo che si possa sempre accedere all'indirizzo base di una gerarchia, mentre un dominio con suffisso viene trasformato dalla funzione di hashing. Nel caso di un tipo *OTIP* non sono previsti domini di base, ma è necessario passare come parametri l'orario attuale (espresso in secondi dalla Unix epoch) e il timeslice previsto dalla configurazione del dominio utilizzando la macro `iothaddr_otiptime`. Il record di risposta di indirizzi *OTIP* specifica inoltre un Time To Live di 1 secondo per prevenire che un indirizzo temporaneo venga messo in una qualche cache da parte degli utilizzatori. È importante notare come la funzione aggiunga RR esclusivamente per i pacchetti richiedenti record AAAA: se una risposta DNS per indirizzi hash contenesse anche record A di indirizzi IPv4, il sistema ricevente potrebbe erroneamente scegliere di utilizzarli al posto dei corretti indirizzi IPv6. Gli indirizzi generati dalla funzione `iothaddr_hash` sono arbi-

trariamente generati e sono dunque considerati amministrati localmente, di conseguenza il settimo e ottavo bit di interfaccia sono messi a 0 in seguito all'hashing.

2.2.4 Forwarding di richieste

Mentre una risposta a un record già presente nella configurazione di VdeDNS può essere costruita immediatamente ricevuta una query, nel caso in cui sia necessario fare forwarding l'attesa della risposta da parte di un master DNS non può essere bloccante. VdeDNS utilizza di conseguenza un sistema asincrono in cui le richieste in forwarding vengono temporaneamente sospese e riprese solo quando sia giunta una risposta. L'implementazione dell'asincronia tra UDP e TCP non ha alcuna differenza sostanziale su VdeDNS, verrà presa come esempio quella con meno overhead di UDP.

Nel momento in cui sia necessario fare forwarding di una richiesta, viene composto e inviato un pacchetto di query al master DNS e inserite nella Request Queue/Table le informazioni relative alla richiesta e al mittente originali, alle eventuali informazioni relative a parametri di *HASH*/*OTIP* e alla scadenza della richiesta. Quando il socket in attesa di risposte riceve un pacchetto, si verifica che l'ID e il query name facciano match con una richiesta in sospeso nella lista di trabocco corrispondente della Request Table e nel caso si utilizzano le informazioni di risposta per comporre e spedire il pacchetto al mittente originale.

La configurazione di VdeDNS permette di impostare più di un master DNS di riferimento per il forwarding, in modo che possa essere disponibile un fallback nel momento che il primo dovesse fallire o rispondere troppo lentamente. Questi DNS vengono consultati in successione nel caso che la richiesta resti sospesa più a lungo di un determinato tempo di timeout, rimuovendo la richiesta scaduta dalla struttura dati e aggiungendone una nuova con un altro ID unico. Questo passaggio richiede quindi che nella Request Queue/Table rimanga salvato il pacchetto di query composto precedentemente in fase di parsing, in modo da poterlo trasmettere trasparentemente al DNS successivo.

Listing 2.4 : manage_req_queue

```c
static void manage_udp_req_queue(){
  struct hashq *current = NULL;
  struct hashq *iter;
  while((iter = next_expired_req(&current)) != NULL){
    struct dnsreq *req = (struct dnsreq*)iter->data;
    printlog(LOG_DEBUG, "Expired UDP Request ID: %d Query: %s\n",
      req->h.id, req->h.qname);
    //free id previousy in use
    free_id(req->h.id);
    //if there are more available dns we query them aswell
    if(qdns[++req->dnsn].sin6_family != 0){
      char origdom[IOTHDNS_MAXNAME];
      struct pktinfo pinfo;

      //generate new id
      pinfo.h = &req->h;
```

```
      pinfo.h->id = get_unique_id();
      req->pktbuf[0] = pinfo.h->id >> 8;
      req->pktbuf[1] = pinfo.h->id;

      pinfo.origdom = origdom;
      pinfo.origid = req->origid;
      strncpy(pinfo.origdom, req->origdom, IOTHDNS_MAXNAME);
      pinfo.type = req->type;
      pinfo.opt = req->opt;
      _fwd_udp_req(req->pktbuf, req->pktlen,
          &req->addr, req->addrlen, &pinfo, req->dnsn);
    }
    free_req(iter);
  }
}
```

La funzione `manage_req_queue` si occupa di cercare richieste scadute all'interno della Request Queue e viene chiamata ogni volta che la funzione di polling dovesse fare timeout o alternativamente, se non dovesse esserci un timeout a causa di continua attività dei file descriptor, in maniera periodica allo scadere di un timer. Per come si è visto essere strutturata la Request Queue, al termine del ciclo `while`, cioè alla prima richiesta non scaduta in coda, si ha la garanzia che le richieste restanti siano ancora valide. Quando viene trovata una richiesta scaduta si verifica che ci siano ancora dei master DNS disponibili e in caso positivo si libera l'ID precedentemente in uso, se ne genera uno nuovo per la richiesta e si inoltra al nuovo DNS, aggiungendo in fondo alla coda la richiesta. La richiesta precedente viene in ogni caso abortita rimuovendola dalla struttura dati per prevenire possibili conflitti.

2.2.5 Implementazione UDP

L'implementazione della connettività DNS per il thread UDP è generalmente molto diretta e adopera l'interfaccia di *libioth* [22] per le chiamate di funzioni di rete. Essendo UDP connectionless, è sufficiente l'uso di due soli socket, uno per le connessioni tra client e VdeDNS e un altro per le connessioni tra VdeDNS e i master DNS. L'intero main loop per questa implementazione fa polling sui due socket sopracitati e non deve fare alcun parsing particolare al buffer ricevuto da una `ioth_recv_from`, poiché da standard ogni datagramma UDP contiene esattamente un pacchetto DNS. [16]

Listing 2.5 : send_udp_ans

```
void send_udp_ans(int fd, unsigned char* buf, ssize_t len,
    struct sockaddr_storage* from, socklen_t fromlen){
  //packet truncation check
  //if a packet is longer than 512 bytes
  //(or larger custom buffer, but this is rare)
  //we send an empty answer with truncation bit on
  if(len > udp_maxbuf){
    struct iothdns_header h;
```

```
    char qname[IOTHDNS_MAXNAME];
    struct iothdns_pkt* pkt = iothdns_get_header(&h, buf, len,
       qname);
    iothdns_free(pkt);
    h.flags |= IOTHDNS_TRUNC;
    pkt = iothdns_put_header(&h);
    ioth_sendto(sfd, iothdns_buf(pkt), iothdns_buflen(pkt), 0,
        (struct sockaddr *) from, fromlen);
  } else {
      ioth_sendto(sfd, buf, len, 0,
          (struct sockaddr *) from, fromlen);
  }
}
```

L'unica vera particolarità dell'implementazione UDP si trova nella funzione `send_udp_ans` chiamata in seguito alla costruzione di un pacchetto di risposta per mandarlo al client di destinazione. Nel caso la dimensione del pacchetto costruito dovesse eccedere il limite di 512 byte (o il limite esteso, configurabile al lancio del programma), allora il protocollo prevede l'invio del pacchetto con bit di troncamento acceso, come accennato in precedenza. Non è precisamente specificato come questo pacchetto debba essere modificato, ma la soluzione naif implementata inizialmente di lasciare invariato il pacchetto, troncarlo alla dimensione limite e accendere il bit di troncamento portava all'invio di un pacchetto malformato che causava errori ai client DNS. Si è scelto in questo caso di imitare il comportamento di alcuni provider DNS principali (Google, Cloudflare), che inviano un pacchetto di risposta con l'header modificato accendendo il bit di troncamento, ma senza includere alcun record di risposta, mandando quindi una risposta vuota.

2.2.6 Implementazione TCP

L'implementazione asincrona di TCP con polling non è immediata. Si richiede ci sia un socket di accettazione per i client, che a sua volta genera nuovi socket per le singole sessioni, e un socket per ogni connessione verso un master DNS. Inoltre, non è garantito che un singolo segmento inviato o ricevuto con TCP corrisponda a un pacchetto completo, è anzi incoraggiato il pipelining per ridurre i tempi di attesa e compensare all'overhead introdotto dal protocollo. [10] Mentre l'invio potenzialmente a segmenti di pacchetti è gestito dal sistema operativo, la ricezione segmentata di pacchetti lungo un flusso richiede un parsing manuale. L'implementazione di questa specifica tramite polling richiede dunque un sistema ben strutturato.

Listing 2.6 : struct conn

```
//basic struct
struct conn {
  int fd;
  uint8_t state;
};

//struct for connections from clients
```

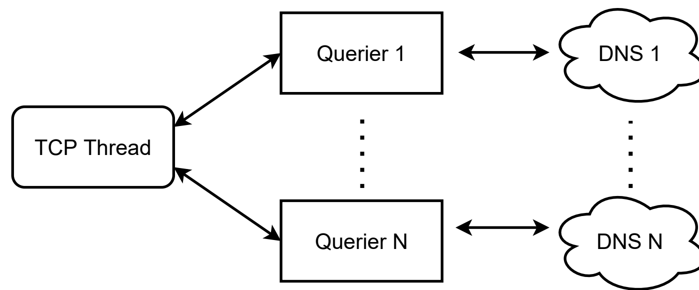


Figura 2.3: Struttura dei querier TCP

```
struct clientconn {
  int fd;
  uint8_t state;
  unsigned char* buf;
  ssize_t buflen;
  uint16_t pktlen;
  struct sockaddr_storage from;
  socklen_t fromlen;
};
```

Come spiegato precedentemente, l'utilizzo della funzione `epoll` in questa situazione è nettamente preferibile per rendere più facilmente gestibile e scalabile la struttura del programma. La `epoll` può restituire una struttura dati arbitraria associata a un file descriptor, che permette di mantenere un'arbitraria mole di dati relativi alle singole connessioni. Facendo uso di questa funzionalità è possibile utilizzare una struttura dati di base `struct conn` contenente il file descriptor e un token rappresentante il tipo di socket (i.e. in ascolto, di un client, di un server. . .) e successivamente espanderla in strutture dati più complesse ed eterogenee, che possono comunque essere confrontate tra loro sui valori di base tramite casting (simulando l'inheritance dell'Object Oriented Programming).

Per gestire il forwarding di richieste tramite TCP permettendo il pipelining sarebbe necessario tenere vive le connessioni verso i master DNS selezionati, quindi fare polling costante su ogni file descriptor per controllare eventuali errori di *HANGUP*, in cui nel caso riaprire una nuova sessione richiamndo `ioth_connect`. Si è scelto di effettuare un'astrazione di questo processo creando un querier thread per ogni master DNS (figura 2.3), a cui si mandino pacchetti di query e da cui si ricevano risposte e che si occupi di effettuare e mantenere una connessione TCP a ogni occorrenza. Così facendo è sufficiente creare un socket per inter-process communication (IPC), usando la system call `socketpair`, verso ogni querier a cui fare forwarding di pacchetti e da cui fare normalmente polling sul thread principale, senza doversi preoccupare di effettuare o tenere vive connessioni o di causare race condition tra thread. Ogni volta che il querier thread riceve un pacchetto controlla se la connessione verso il DNS sia attiva o meno e in caso negativo si occupa di richiamare `ioth_connect` per poter inviare la query.

Il socket in ascolto per nuovi client è creato con l'opzione `SOCK_NONBLOCK`, per prevenire un blocco del thread principale nel caso una connessione cada nel momento in cui viene ricevuto il segnale di polling. [21] Le connessioni al thread TCP sono gestite con l'uso

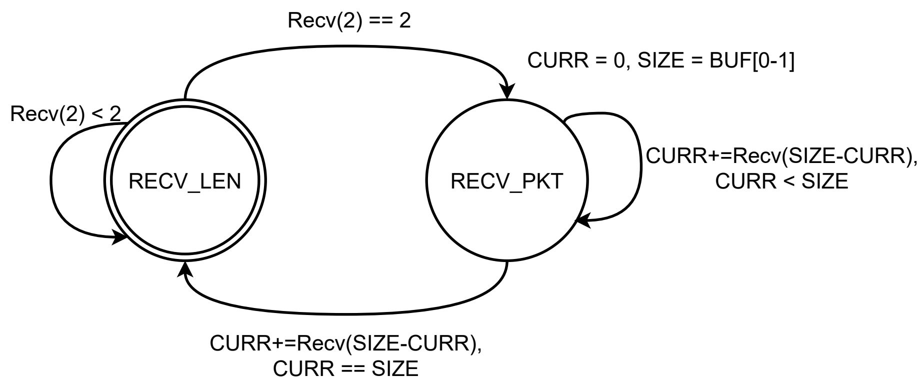


Figura 2.4: Automa a stati per ricezione TCP

della struttura dati associata al polling vista in precedenza e con una TCP File Descriptor Queue/Table (TCPFD Queue/Table) per gestire eventuali timeout di connessioni. La TCPFD Queue/Table è implementata sullo stesso template della Request Queue/Table. Quando avviene un evento da un file descriptor si accede all'elemento in tempo costante dalla Table e lo si sposta in fondo alla Queue (essendo questa implementata con sentinella, anche questa operazione è eseguita in tempo costante). Nello stesso modo in cui viene controllata la Request Queue, si controlla periodicamente anche la TCPFD Queue per rimuovere elementi scaduti: in questo caso oltre alla rimozione avviene una chiusura del file descriptor.

Ricevere correttamente uno stream da un client nel protocollo DNS implica dover fare parsing di ogni buffer ricevuto per ricomporre pacchetti. All'interno di un solo buffer potrebbero trovarsi più pacchetti e l'ultimo di questi potrebbe essere solo un frammento di un pacchetto completo. Ognuno di questi è preceduto da due byte indicanti la sua lunghezza, allo scopo di permettere di estrarre il corretto numero di byte dal buffer. In VdeDNS si utilizza un meccanismo a stati (figura 2.4): ogni struttura dati associata a un socket di un client (o di un server DNS, nel caso di ricezione risposte) si trova in stato `RECV_LEN` o `RECV_PKT`. Alla creazione del socket è impostato lo stato di partenza `RECV_LEN`, appena si ricevono i due byte di lunghezza questa viene memorizzata e si passa in stato `RECV_PKT`, in cui viene ricevuto il buffer del pacchetto fino alla lunghezza preannunciata. Una volta ricevuto il numero preannunciato di byte, si ritorna in stato `RECV_LEN` per passare alla lettura del pacchetto successivo. Questo processo è implementato esclusivamente tramite polling e permette di gestire correttamente buffer con pacchetti multipli e/o frammentati.

Listing 2.7 : make_tcp_pkt

```
//allocates tcp dns packet from udp packet
static void *make_tcp_pkt(void* buf, ssize_t *len){
  unsigned char* tcpbuf = malloc(*len+2);
  tcpbuf[0] = (*len) >> 8;
  tcpbuf[1] = (*len);
  memcpy(tcpbuf+2, buf, *len);
  *len += 2;
```

```
  return (void*)tcpbuf;
}
```

Quando la ricezione di un pacchetto è completata questo viene mandato senza i primi due byte di lunghezza al parser DNS, che lo interpreta esattamente come un pacchetto UDP generico. La funzione di risposta di TCP riceve quindi un pacchetto non immediatamente idoneo, che però trasforma banalmente aggiungendoci in testa la sua lunghezza in due byte tramite la funzione `make_tcp_pkt`.

2.3 Sperimentazione

2.3.1 Setup e risoluzione indirizzi

Il funzionamento di VdeDNS può essere sottoposto a test in maniera molto diretta, richiedendo la risoluzione di domini locali o esterni e verificando che le risposte siano in linea con le aspettative.

Per prima cosa si modifica un file di configurazione aggiungendo domini da risolvere come *HASH* e *OTIP*, per poi aggiungere alcuni di questi ai record locali.

Listing 2.8 : Configurazione VdeDNS

```
rules = {
    hash = (
        "hashdomain.vde",
        "debian.org"
    );
    otip = (
        {
            dom="otipdomain.vde";
            pswd="secret";
            time=32;
        }
    );
};
records = (
    {
        dom="hashdomain.vde";
        ip6=("fc00:4242::17");
    },
    {
        dom="otipdomain.vde";
        ip6=("fc00:4242::13");
    },
    {
        dom="vdednstest.vde";
        ip4=("10.9.8.7");
        ip6=("fc00:a987::1");
    }
```

```
);
```

Si avvia VdeDNS localmente includendo la configurazione posta sopra per le regole di risoluzione domini e di record locali.

```
sudo vdedns -c configfile.cfg -b 127.0.0.1
```

Poi si modificano le impostazioni DNS di sistema in modo che le richieste vengano fatte verso localhost (nel caso di sistemi unix-like tipicamente aggiungendo in cima al file */etc/resolv.conf* la stringa `nameserver 127.0.0.1`).

Si eseguono a questo punto richieste tramite la utility `host`.

```
$ host hashdomain.vde
hashdomain.vde has IPv6 address fc00:4242::17
$ host sub.hashdomain.vde
sub.hashdomain.vde has IPv6 address fc00:4242::a4bd:49d:a6e5:7fcd
$ host fc00:4242::a4bd:49d:a6e5:7fcd
d.c.f.7.5.e.6.a.d.9.4.0.d.b.4.a.0.0.0.0.0.0.0.0.2.4.2.4.0.0.c.f.
   ip6.arpa domain name pointer sub.hashdomain.vde.
$ host debian.org
debian.org has IPv6 address 2603:400a:ffff:bb8::801f:3e
debian.org has IPv6 address 2001:67c:2564:a119::77
debian.org has IPv6 address 2001:4f8:1:c::15
$ host mydom.debian.org
mydom.debian.org has IPv6 address 2001:4f8:1:c:e0b9:8d8b:21e:2792
mydom.debian.org has IPv6 address 2603:400a:ffff:bb8:e0b9:8d8b
   :8201:27b9
mydom.debian.org has IPv6 address 2001:67c:2564:a119:e0b9:8d8b:21
   e:27f0
$ host otipdomain.vde
otipdomain.vde has IPv6 address fc00:4242::40c1:4a67:f5a2:e60
$ host fc00:4242::40c1:4a67:f5a2:e60
Host 0.6.e.0.2.a.5.f.7.6.a.4.1.c.0.4.0.0.0.0.0.0.0.0.2.4.2.4.0.0.
   c.f.ip6.arpa not found: 3(NXDOMAIN)
$ host vdednstest.vde
vdednstest.vde has address 10.9.8.7
vdednstest.vde has IPv6 address fc00:a987::1
```

Si può osservare come da aspettative che:

- Una richiesta a un dominio *HASH* locale senza sottodominio ritorna l'indirizzo base IPv6
- Una richiesta a un dominio *HASH* locale con sottodominio ritorna l'indirizzo hash IPv6
- Una richiesta a un indirizzo *HASH* locale ritorna il dominio salvato in seguito alla richiesta precedente
- Una richiesta a un dominio *HASH* esterno con più indirizzi IPv6 senza sottodominio ritorna tutti gli indirizzi base IPv6 ottenuti tramite forwarding (escludendo le risposte IPv4)

- Una richiesta a un dominio *HASH* esterno con più indirizzi IPv6 con sottodominio ritorna tutti gli indirizzi hash IPv6 ottenuti tramite forwarding (escludendo le risposte IPv4)

- Una richiesta a un dominio *OTIP* locale ritorna l'indirizzo hash IPv6

- Una richiesta a un indirizzo *OTIP* locale non ritorna alcun dominio corrispondente (non si possono fare associazioni con indirizzi temporanei)

- Una richiesta a un dominio comune locale ritorna tutti gli indirizzi impostati nei record

2.3.2 Performance di VdeDNS

Grande parte dello studio riguardante lo sviluppo di VdeDNS è dedicato al renderlo veloce e leggero, in modo che introduca meno overhead possibile quando in uso. Mentre le risoluzioni di record locali sono ovviamente molto veloci, la parte interessante sono le risoluzioni di record per cui sia necessario richiedere forwarding ai master DNS.

Come esperimento si vuole mettere alla prova la capacità di VdeDNS di gestire richieste contemporanee non locali. Il test consiste nell'effettuare 100 richieste DNS verso vari domini lanciando allo stesso tempo un thread per ogni richiesta e attendendone il responso. Il test si conclude quando ogni thread ha ricevuto la sua risposta ed è quindi terminato. Utilizzando il programma C `threadreq`, che prende come argomento un numero di richieste DNS da effettuare in contemporanea tramite threading, si scrive uno script *bash* che misuri un numero di esecuzioni, ne stampi il tempo in millisecondi e ne misuri la media.

Listing 2.9 : Speedtest Bash script

```bash
#!/usr/bin/bash

N=100
SUM=0

#https://stackoverflow.com/questions/16959337/usr-bin-time-format
    -output-elapsed-time-in-milliseconds
for i in $(seq 1 $N); do
    ts=$(date +%s%N);
    ./threadreq 100 > /dev/null;
    tt=$((($(date +%s%N) - $ts)/1000000));
    SUM=$(($SUM+$tt));
    echo $tt;
    #prevent DNS flood protection
    sleep 0.5
done;

echo "Average␣$(($SUM/$N))"
```

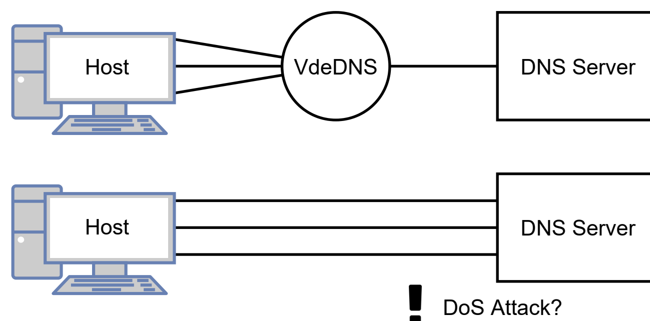


Figura 2.5: Stream TCP DNS multipli

Si esegue lo script tentando di emarginare il più possibile i risultati outlier (di solito richieste dell'ordine di 500+ millisecondi) dovuti a delay nella rete causati probabilmente da meccanismi di protezione del server DNS. Il test è eseguito con e senza VdeDNS attivo, prima con UDP e poi con TCP, utilizzando il servizio DNS di Cloudflare (1.1.1.1) prima come server DNS diretto e poi come master DNS del proxy. L'ambiente di esecuzione è un Dell G5 5587 con processore Intel Core i7-8750H, 16 GB di ram e sistema operativo Debian 11 (Bullseye), con rete cablata e connessione a fibra Gigabit. La configurazione di VdeDNS in questo caso include 100 domini *HASH*, 100 domini *OTIP* e 100 record locali su cui il proxy debba iterare, nel tentativo di aggiungere una quantità realistica di overhead.

Protocollo	No Proxy	VdeDNS
UDP	81ms±31ms	84ms±28ms
TCP	98ms±24ms	83ms±29ms

Tabella 2.1: Test di velocità di risoluzione DNS con e senza proxy

I risultati del test non sono completamente in linea con le aspettative iniziali (tabella 2.1). Per UDP il risultato è come aspettato, l'overhead aggiunto dal proxy è probabilmente trascurabile e la differenza di delay è minimale e possibilmente dovuta all'imperfezione dei sistemi di rete reali. Sorprendentemente, la performance del proxy nel test risulta evidentemente migliore di una diretta richiesta di sistema, mentre l'aspettativa era valutare al contrario un eventuale delay dovuto all'overhead del parsing del proxy. Non solo, ma durante i test le richieste TCP dirette mostrano molti più casi outlier in cui un'esecuzione di `threadreq 100` può impiegare anche più di un secondo.

Indagando sul fenomeno si ricorda che l'implementazione TCP di VdeDNS prevede che le richieste a un master DNS siano effettuate su un unico query thread, che apre e tiene viva una singola connessione per volta. Conseguentemente, tutte le richieste DNS tramite TCP inviate al proxy vengono inoltrare al master DNS remoto su un unico stream, mentre le richieste dal sistema inviate direttamente tramite `getaddrinfo` aprono ciascuna un nuovo stream diretto (figura 2.5). L'apertura di uno stream TCP richiede una negoziazione tramite il *TCP handshake*, che aggiunge inevitabilmente overhead iniziale alla comunicazione. Utilizzando VdeDNS come proxy locale, vengono eseguiti 100 handshake paralleli in rete locale e un singolo handshake verso un server remoto, che risultano essere più veloci di 100 handshake paralleli verso un server remoto e aggiungono

dunque un overhead visibile nel test. In aggiunta, si possono anche spiegare l'elevato numero di casi outlier ottenuti nel test TCP: i server DNS principali e più utilizzati (come in questo caso è quello di Cloudflare) hanno diverse protezioni contro attacchi di Denial Of Service (DoS), in questo caso particolare viene probabilmente imposto un limite sul numero di connessioni TCP aperte contemporaneamente da uno stesso indirizzo IP (come effettivamente suggerito dall'RFC7766 [10]). Possiamo quindi osservare come l'uso di VdeDNS per la gestione di un grande numero di richieste TCP parallele possa avere un impatto positivo sui tempi di risoluzione.

Capitolo 3

Tool di supporto per OTIP

3.1 Finalità e descrizione

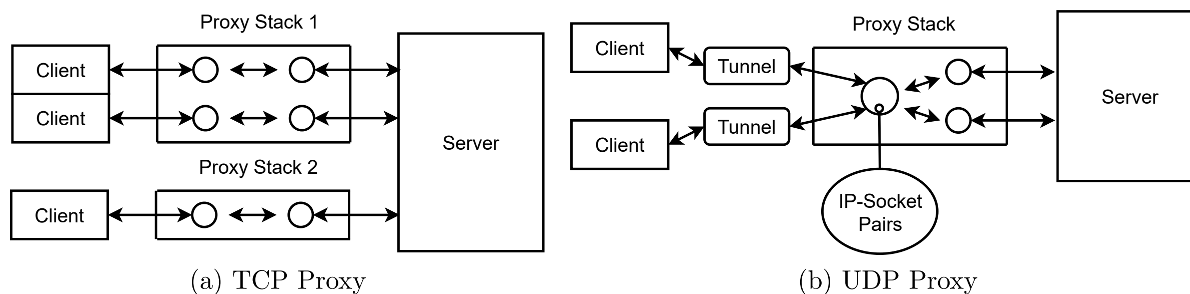
Come descritto precedentemente, il funzionamento di sistemi di sicurezza basati su indirizzi IP temporanei richiede specifiche non presenti nei sistemi attuali. Dal lato del client è necessario computare un indirizzo temporaneo per ogni sessione di connessione, dal lato server è necessario accettare connessioni (o singoli datagrammi) solo sull'indirizzo corrente: come precedentemente discusso, possono essere messi a disposizione tool di supporto per abilitare all'uso di *OTIP* servizi generici non *OTIP-aware*. Tuttavia, non è ovvio che i client e i server, anche se utilizzati in combinazione con tool appropriati, siano predisposti a un funzionamento corretto del meccanismo. Questa sezione vuole discutere del tentativo di implementazione e del testing di alcuni tool a scopo sperimentativo per l'utilizzo delle feature *OTIP*.

VdeDNS risolve il problema della computazione di indirizzi di cui un client necessita, effettuando risoluzioni DNS. Purtroppo, dall'altro lato, un servizio generico non è un server a IP variabile e sarebbe poco realistico riscrivere ogni programma server per adattarlo a questo sistema. Si è infatti precedentemente ipotizzato l'utilizzo di un reverse proxy che gestisca il cambio di indirizzi, allo scopo di abilitare l'uso di *OTIP* su un server di un qualche servizio. Interponendo infatti tra una comunicazione di client e server di servizi generici un reverse proxy a indirizzo variabile, dovrebbe essere possibile emulare il concetto di un reale server *OTIP*. Questo reverse proxy può essere implementato sia per connessioni TCP, che per connessioni UDP, anche se con accorgimenti differenti.

3.1.1 OTIP TCP Proxy

Il punto focale della tecnica di emulazione di un server a indirizzo variabile è il poter adoperare la API fornita da *libioth* per creare e distruggere stack di rete a runtime. Ogni stack è configurato, all'interno della rete virtuale, con un'interfaccia di cui indirizzo IP corrisponde a quello computato per il timeslice in cui è stato generato. Avendo a disposizione lo stack, il proxy può crearvi sopra un nuovo socket da mettere in ascolto, aggiungendo effettivamente un nuovo indirizzo IP di accesso al proxy.

Nell'istanza di un servizio TCP (figura 3.1a), come spiegato in precedenza, si possono accettare nuove connessioni finché uno stack si trovi entro il proprio timeslice e solo al termine chiudere il socket in ascolto, lasciando tuttavia aperte le sessioni già in corso.



(a) TCP Proxy (b) UDP Proxy

Figura 3.1: OTIP Proxy

Finché sono presenti connessioni attive, lo stack opera da tramite tra i due stream di dati client-proxy e proxy-server; nel momento in cui client o server riaggancino la connessione, se questa era l'ultima attiva rimasta e il timeslice fosse terminato allora lo stack può essere definitivamente eliminato. Questo metodo permette di creare l'astrazione di un server TCP a indirizzo variabile tramite l'utilizzo di stack di rete virtuali.

3.1.2 OTIP UDP Proxy e Tunnel

Essendo UDP connectionless, il funzionamento generale, ad eccezione della creazione e distruzione di stack di rete per aggiungere e rimuovere indirizzi, differisce da TCP (figura 3.1b). Per creare collegamenti distinti tra ogni singolo client e il server si creano socket da associare a una coppia indirizzo IP e porta di ritorno, in modo che a ogni risposta da parte del server il proxy sia in grado di inoltrare i datagrammi al client corretto.

Purtroppo non vi è come in TCP il modo di controllare quando una sessione sia terminata (a meno che non si tenti di simulare uno stream con sistemi di timeout) e quindi non è possibile lasciare aperto lo stack finché siano rimaste sessioni attive. Di conseguenza un client UDP, anche facendo uso di un risolutore di indirizzi *OTIP*, dovrebbe essere consapevole del sistema e modificare l'indirizzo di invio anche durante una sessione. È possibile riprodurre questo processo facendo uso di un tunnel: in questo modo un processo generico può comunicare normalmente con un tunnel a indirizzo statico, mentre il tunnel si occupa di modificare (manualmente computando o richiedendo una risoluzione specializzata) l'indirizzo a cui spedire i datagrammi ricevuti.

3.2 Implementazione

Le implementazioni dei tool proposti in seguito sono prototipi creati principalmente a scopo esemplificativo e sono conseguentemente molto limitati e lontani dall'essere considerabili pronti a un uso reale. In tutte le implementazioni gli indirizzi hash temporanei sono computati all'interno del programma, ma un'alternativa forse migliore e più adatta alla standardizzazione sarebbe che gli indirizzi fossero richiesti tramite DNS a un risolutore unico, come ad esempio VdeDNS. In ogni caso, allo stesso tempo, sarebbe comunque indispensabile tenere traccia della durata di un timeslice di un indirizzo, poiché l'alternativa sarebbe effettuare continue chiamate DNS per verificarne la persistenza.

3.2.1 OTIP TCP Proxy

Il proxy per TCP è implementato come un thread server: ogni nuovo stack è lanciato su un diverso thread e ogni nuova connessione aperta verso uno stack genera a sua volta un thread dedicato. Una volta creato il thread dedicato di un client, viene aperto un ulteriore stream tra proxy e server e reindirizzati i segmenti dal client al server e viceversa; quando uno dei due riaggancia la connessione, il thread e i socket vengono chiusi.

Listing 3.1 : TCP Proxy stack loop

```
//will accept connections until time limit and then only close
   after all tcp connections are down
while((time(NULL) - start_time) < (timeslice+offset)){
  //non-blocking socket just in case accept is left pending
  if(poll(fd, 1, 1000) > 0){
    if((client_fd = ioth_accept(proxy_fd, NULL, NULL)) < 0){
      if(errno == EAGAIN || errno == EWOULDBLOCK){
        continue;
      } else break;
    }
    /*INIT THREAD ARGS
    . . .
    */
    pthread_mutex_lock(&lock);
    running_clients++;
    pthread_mutex_unlock(&lock);
  }
}
//stack is open until last client disconnects
while(running_clients > 0){
    sleep(1);
}
ioth_close(proxy_fd);
ioth_delstack(mystack);
pthread_exit(NULL);
```

Il thread relativo allo stack resta in ascolto di connessioni fintanto che il suo timeslice totale (comprensivo di un offset per poter sostenere un certo margine di errore di calcolo dell'indirizzo) è valido. Una volta concluso il timeslice, il thread resta in attesa che i client ancora connessi terminino la loro connessione prima di eliminare lo stack e ritornare. Questo controllo è gestito da una variabile `volatile int running_clients`, che facendo uso di lock viene incrementata a ogni nuova connessione e ridotta a ogni disconnessione rispettivamente nei thread dello stack e dei client.

3.2.2 OTIP UDP Proxy

Il proxy UDP similmente lancia un nuovo thread per ogni stack creato, ma le connessioni al suo interno sono gestite tramite `epoll` con un sistema molto simile a quello adottato sul thread TCP di VdeDNS. Come nel caso precedente, è idoneo specificare un offset per

permettere che due stack si sovrappongano per un margine di tempo sufficiente a evitare errori di coordinazione temporale, ma questa volta lo stack non resta aperto nel caso vi sia una sessione in corso.

Listing 3.2 : UDP Proxy poll event

```
if(EFD(events[i]) == proxy_fd){
//FROM CLIENT
  char buf[BUFSIZE];
  struct sockaddr_in6 fromaddr;
  socklen_t fromlen = sizeof(fromaddr);
  n_bytes = ioth_recvfrom(proxy_fd, buf, BUFSIZE, 0, (struct
     sockaddr*)&fromaddr, &fromlen);
  struct hashq* el;
  struct udp_conn* conn;
  if((el = get_conn((struct sockaddr_in6*)&fromaddr)) != NULL){
  //socket is already communicating
    conn = (struct udp_conn*)el->data;
    update_conn((struct sockaddr_in6*)&fromaddr);
  } else {
  //new communication
    conn = malloc(sizeof(struct udp_conn));
    conn->fd = ioth_msocket(pstack, AF_INET6, SOCK_DGRAM, 0);
    conn->addr = *((struct sockaddr_in6*)&fromaddr);
    conn->addrlen = fromlen;
    event.data.ptr = (void*)conn;
    epoll_ctl(efd, EPOLL_CTL_ADD, conn->fd, &event);
    add_conn(conn);
  }
  ioth_sendto(conn->fd, buf, n_bytes, 0, saddr->ai_addr, saddr->
     ai_addrlen);
} else {
//FROM SERVER
  struct udp_conn* conn = (struct udp_conn*)events[i].data.ptr;
  char buf[BUFSIZE];
  n_bytes = ioth_recv(conn->fd, buf, BUFSIZE, 0);
  ioth_sendto(proxy_fd, buf, n_bytes, 0, (struct sockaddr*)&conn
     ->addr, conn->addrlen);
}
```

Facendo uso di una struttura dati Table/Queue implementata sopra a quelle viste precedentemente, per ogni datagramma ricevuto da un client sul socket in ascolto si controlla se la corrispondente coppia indirizzo e porta risulti presente: in caso positivo si recupera un socket creato precedentemente per la sessione, altrimenti ne viene creato uno nuovo e aggiunte le informazioni nella struttura. Ogni socket UDP creato in questo modo è controllato dinamicamente in polling e associato alla struttura `struct sockaddr_in6` del mittente originale, quando da uno di questi venga ricevuto un messaggio dal server si utilizza il socket in ascolto per farlo inoltrare al client destinatario usando le informazioni salvate. La pulizia della struttura dati avviene con un sistema di timeout in cui vengono

eliminati i socket inattivi scorrendo la coda, esattamente come in VdeDNS. La struttura dati è globale per il programma, conseguentemente i socket di comunicazione con il server sopravvivono ai cambi di stack e un client UDP può dunque effettuare una comunicazione prolungata lungo la durata di più timeslice (a patto ovviamente di mandare i datagrammi all'indirizzo aggiornato).

3.2.3 OTIP UDP Tunnel

L'implementazione del tunnel UDP proposta è estremamente semplice, anche se un po' limitata. L'unico compito del tunnel è mediare la trasmissione di pacchetti tra client e server *OTIP*, facendo in modo che l'indirizzo IP del server resti aggiornato nel tempo.

Listing 3.3 : UDP tunnel loop

```
  for(;;){
    if((time(NULL)%timeslice == 0)){
      saddr.sin6_addr = byteaddr;
      iothaddr_hash(&saddr.sin6_addr, fqdn, pswd,
        iothaddr_otiptime(timeslice, 0));
    }
        poll(fds, 2, 800);
        if(fds[0].revents){
    //MESSAGE FROM CLIENT
          caddr_size = sizeof(caddr);
          n_bytes = recvfrom(cfd, buf, BUFSIZE, 0,
        (struct sockaddr*)&caddr, &caddr_size);
          sendto(sfd, buf, n_bytes, 0,
        (struct sockaddr*)&saddr, sizeof(struct sockaddr_in6));
        }
        if(fds[1].revents){
    //MESSAGE FROM SERVER
          caddr_size = sizeof(caddr);
          n_bytes = recv(sfd, buf, BUFSIZE, 0);
          sendto(cfd, buf, n_bytes, 0, (struct sockaddr*)&caddr,
          sizeof(struct sockaddr_in6);
    }
}
```

All'interno del loop principale, si modifica l'indirizzo IP associato alla struttura `struct sockaddr_in6 saddr` ogni volta che sia terminato il timeslice dell'indirizzo precedente. Questa struttura è poi utilizzata in `ioth_sendto` per spedire i datagrammi ricevuti dal client al corretto indirizzo del server.

È evidente che per prima cosa il tunnel proposto non sia pensato per servire più client contemporaneamente, dato che l'invio della risposta dal server al client è indirizzata all'indirizzo e porta dell'ultimo datagramma ricevuto. Inoltre potrebbero risultare problemi di desincronizzazione, che dovrebbero però essere mitigati da un margine di offset di accessibilità dell'indirizzo lato server.

3.3 Sperimentazione

Si possono provare ad abilitare alcuni servizi basilari a un utilizzo con *OTIP* come test di funzionamento e osservare il comportamento dell'interazione client-server. Tutti i test vengono eseguiti localmente con ausilio di reti virtuali VDE.

Come setup basilare per ogni test si crea un'interfaccia TAP e vi si assegnano un indirizzo IPv4 e un indirizzo IPv6, poi si lancia `vde_plug` come demone creando uno switch VDE in */tmp/sw*, che verrà usato con Virtual Network Locator per gli stack virtuali:

```
$ sudo ip tuntap add mode tap tap0
$ sudo ip addr add 10.0.0.254/24 dev tap0
$ sudo ip addr add fc00:fede::42/64 dev tap0
$ sudo ip link set tap0 up
$ vde_plug -d switch:///tmp/sw tap://tap0
```

I server dei servizi vengono aperti su localhost, mentre il reverse proxy su una rete virtuale *vdens* [27] all'indirizzo 10.0.0.1 e fc00:fede::43. Su una nuova shell si lancia quindi:

```
$ vdens vde:///tmp/sw
$ ip addr add 10.0.0.1/24 dev vde0
$ ip addr add fc00:fede::43/64 dev vde0
$ ip link set vde0 up
```

3.3.1 SSH

Il primo servizio su cui sperimentare è Secure Shell (SSH), che permette connessioni TCP per interagire con macchine remote. SSH esegue un singolo handshake TCP e una singola sessione non termina fino a che client o server non riaggancino la connessione, quindi risulta essere un buon candidato per un test iniziale.

Si configura SSH dal file */etc/ssh/sshd_config* sulla macchina virtuale in modo che sia in ascolto esclusivamente sull'interfaccia tap0:

```
Port 22
AddressFamily any
ListenAddress 10.0.0.254
ListenAddress fc00:fede::42
```

Il file di configurazione *ssh.otip* per lanciare il reverse proxy per servire SSH nel test è:

```
#otip server domain name
fqdn = sshtest.vde
#target server address
addr = 10.0.0.254
#target server port
port = 22
#otip stack base address
baseaddr = fc00:fede::1234
#vde vnl
vnl = vde:///tmp/sw
```

```
#otip stack gateway
gw = 10.0.0.254
#otip info
pswd = secret
timeslice = 32
offset = 8
```

A questo punto si può lanciare il proxy sulla rete VDE su porta 2222 con `./proxy 2222 ssh.otip`.

Per connettersi tramite dominio si aggiunge alla configurazione di VdeDNS il profilo *OTIP* impostato per SSHY:

```
rules = {
    ...
    otip = (
        {
            dom="sshtest.vde";
            pswd="secret";
            time=32;
        },
    ...
records = (
    {
        dom="sshtest.vde";
        ip6=("fc00:fede::1234");
    },
    ...
```

Una volta avviato VdeDNS e dopo averlo specificato come server DNS primario, si può tentare una connessione SSH a localhost passando attraverso il reverse proxy:

```
$ ssh sshtest.vde -p 2222
```

La connessione SSH va a buon fine e non ha interruzioni nemmeno al termine del timeslice, dal momento che il proxy ha ancora una connessione in corso sullo stack. Anche effettuare multiple connessioni in momenti differenti non ha effetto sulla stabilità del servizio. Dal test eseguito risulta quindi che SSH sia un servizio utilizzabile senza difficoltà con il setup di *OTIP* proposto.

3.3.2 Web Server

Un altro servizio su cui eseguire una semplice sperimentazione è un Web Server. Un Web Server ha qualche complicazione aggiuntiva rispetto a SSH: in primo luogo il client principale è un Web Browser, che è un'entità tipicamente complessa e con alcune scelte implementative opache all'utilizzatore, e in secondo luogo una sessione di "browsing" non è continua, ma prevede invece una connessione per ogni invio di uno o più file.

Come prima, si configura un file per il proxy:

```
#otip server domain name
fqdn = sshtest.vde
#target server address
```

```
addr = 10.0.0.254
#target server port
port = 22
#otip stack base address
baseaddr = fc00:fede::1234
#vde vnl
vnl = vde:///tmp/sw
#otip stack gateway
gw = 10.0.0.254
#otip info
pswd = secret
timeslice = 32
offset = 8
```

E si aggiunge la relativa configurazione di VdeDNS:

```
rules = {
    ...
    otip = (
        {
            dom="webtest.vde";
            pswd="password";
            time=32;
        },
    ...
records = (
    {
        dom="webtest.vde";
        ip6=("fc00:fede::8080");
    },
    ...
```

Il test di navigazione è effettuato su Debian 11 (Bullseye) con Web browser Firefox (versione 78.11.0esr) e Chromium (versione 90.0.4430.212). Per la riproduzione del test in locale è da notare che Chromium presenta un bug relativo alla risoluzione IPv6: se la macchina non ha a disposizione connettività globale IPv6, Chromium si rifiuta di richiedere dei record DNS AAAA impedendo risoluzioni IPv6 locali.

Pagina Web statica

Si prova per prima cosa l'efficacia di navigazione su pagine Web statiche. Si fa uso di un semplice server Python, facendo bind su tap0 e porta 8888 in una directory a scelta:

```
$ python3 -m http.server --bind 10.0.0.254 8888
Serving HTTP on 10.0.0.254 port 8888 (http://10.0.0.254:8888/)
   ...
```

Su Firefox la connessione ha successo e la navigazione avviene correttamente durante la durata del timeslice. Terminato il timeslice, tuttavia, proseguire la navigazione muovendosi per collegamenti ipertestuali risulta impossibile e la connessione dà timeout. Aggiornare manualmente la pagina consente di riprendere a navigare entro il nuovo timeslice. La ragione dietro al problema sorto è riconducibile al fatto che ogni interazione

con un collegamento ipertestuale necessita la riapertura di una connessione TCP, che non sarebbe un problema se fosse preceduta da una nuova chiamata DNS. Purtroppo il browser per ottimizzare il traffico di dati mantiene in cache l'indirizzo risolto dopo la prima chiamata DNS e lo riutilizza finché non venga forzato un aggiornamento della pagina, impedendo l'accesso all'indirizzo *OTIP* più aggiornato.

Su Chromium il comportamento è estremamente simile, ma forzare l'aggiornamento della pagina non sempre genera un'ulteriore chiamata DNS come su Firefox. Talvolta pare sia necessario attendere il timeout completo di connessione prima di richiedere nuovamente la pagina.

Pagina Web AJAX

Successivamente si osserva il comportamento del sistema di *OTIP* in congiunzione con una pagina Web facente uso di Asynchronous JavaScript and XML (AJAX). Si usa una pagina HTML che ogni N secondi fa una richiesta al server per un numero casuale e lo visualizza, mentre il server semplicemente serve la pagina come index e risponde al metodo GET */rng*.

La situazione in questa istanza si complica. Su entrambi i browser, per un valore $N \lessapprox 4$, pare che la pagina Web riesca a funzionare correttamente e che il numero venga correttamente aggiornato a ogni chiamata HTTP. Per un valore $N \gtrapprox 5$, il comportamento è meno prevedibile e la pagina occasionalmente smette di aggiornare il numero per un limitato periodo di tempo, per poi ricominciare.

Osservando l'interazione nel dettaglio, si può notare che con un valore di N basso la sessione di connessione HTTP al server non cade mai (vi è solo quindi un handshake TCP iniziale), mentre con un valore più alto di N la sessione termina e viene riaperta a ogni richiesta. Questo comportamento si spiega andando a investigare gli header delle richieste: sia la richiesta GET per ottenere la pagina sia le successive per ottenere il numero contengono l'header *Keep-Alive*, che chiede al server di mantenere viva la connessione TCP per un determinato periodo di tempo. Conseguentemente, se la frequenza delle richieste AJAX è inferiore al tempo di *Keep-Alive* richiesto, il server, se supporta la specifica, manterrà la connessione continuamente attiva senza lasciarla cadere.

Nel caso di N più alto di una sessione *Keep-Alive*, quello che succede è che finché il timeslice resta valido una connessione successiva continuerà ad andare a buon fine, ma terminato il timeslice il server smetterà di essere raggiungibile. Dopo un breve periodo di di irraggiungibilità, i browser eseguono una nuova richiesta DNS e aggiornano l'indirizzo, permettendo a chiamate successive di raggiungere nuovamente il server per la durata del timeslice.

Per *OTIP*, questo implica che uno stream unico HTTP mai interrotto, come nel caso di un N basso, sia correttamente funzionante senza rischio di incorrere nei problemi di caching causati dall'uso di più connessioni TCP consecutive, mentre nel caso contrario il comportamento potrebbe risultare troppo imprevedibile per un uso reale.

3.3.3 Echo server UDP

Il test UDP è eseguito con un echo server creato utilizzando l'utility `netcat`.

Come nei casi precedenti, si configurano il reverse proxy e VdeDNS:

```
#otip server domain name
fqdn = udp.vde
#target server address
addr = 10.0.0.254
#target server port
port = 4444
#otip stack base address
baseaddr = fc00:fede::1234
#vde vnl
vnl = vde:///tmp/sw
#otip stack gateway
gw = 10.0.0.254
#otip info
pswd = secret
timeslice = 16
offset = 4
```

```
rules = {
    ...
    otip = (
        {
            dom="udp.vde";
            pswd="password";
            time=16;
        },
    ...
records = (
    {
        dom="udp.vde";
        ip6=("fc00:fede::1234");
    },
    ...
```

In questo caso però il proxy è eseguito con l'opzione per utilizzare il protocollo UDP:

```
$ ./proxy -u 2000 udp.otip
```

E si lancia un server ncat con:

```
$ ncat -e /bin/cat -k -u -l 4444
```

Essendo questa istanza UDP, è anche necessario adoperare il tunnel, che viene aperto di default su localhost porta 4242:

```
$ ./tunnel fc00:fede::1234 2000 udp.vde secret --timeslice 16
```

Una volta completato il setup, si può effettuare una connessione UDP adoperando nuovamente netcat e collegandosi al tunnel:

```
$ nc -u 127.0.0.1 4242
Hello World!
Hello World!
```

Essendo UDP connection-less non si può ovviamente parlare di "connessione", ma la sessione *netcat* verso l'echo server sopravvive senza difficoltà ai cambiamenti di stack grazie alla mediazione del tunnel, che cambia gli indirizzi astraendo completamente l'azione al client reale. Per quanto l'utilizzo del tunnel rimuova ulteriormente trasparenza alle comunicazioni, facendosi carico dell'invio di datagrammi a livello trasporto dovrebbe essere in grado di supportare ogni generico servizio UDP.

Conclusione e sviluppi futuri

Nell'operato si sono approfonditi gli aspetti più recenti delle tecnologie relative a IPv6, DNS e virtualizzazione di reti allo scopo di utilizzarli nello sviluppo di strumenti appropriati all'abilitazione dell'uso delle nuove feature proposte per gli indirizzi hash IPv6. Parte di questi strumenti era già disponibile in qualche forma, ma le loro implementazioni erano parzialmente limitate da uno stato dell'arte datato, in particolar modo relativamente alla virtualizzazione di stack di rete, che ha avuto evoluzioni molto recenti. Lo sviluppo è stato poi accompagnato da un'analisi di base dell'uso degli artefatti prodotti, con l'obiettivo di verificare se i risultati ottenuti fossero in linea con le aspettative e se ci fossero eventuali limitazioni dovute all'interazione con sistemi reali.

Si è osservato come VdeDNS si comporti correttamente nella risoluzione delle possibili varianti di query per domini specializzati, sia dirette sia inverse. La performance nello stress test eseguito ha dato risultati positivi, mostrando nella particolare istanza di un elevato numero di richieste parallele un overhead trascurabile per UDP e un miglioramento della velocità per TCP. Inoltre VdeDNS rispetta i requisiti basilari di funzionalità imposti dallo stato dell'arte dei proxy DNS, permettendone l'uso in un ampio spettro di situazioni.

Il reverse proxy TCP/UDP e il tunnel UDP per l'abilitazione all'uso di OTIP hanno avuto risultati meno stabili del proxy DNS, ma hanno comunque mostrato come l'idea della protezione di servizi tramite server a IP variabile sia attuabile. Nonostante il test su UDP sia stato semplice e forse un po' superficiale, concettualmente l'uso del tunnel dovrebbe renderlo un metodo stabile anche se a costo della trasparenza che si avrebbe altrimenti con uno scambio di datagrammi end-to-end. Il metodo adottato per l'implementazione di TCP permette una connessione più trasparente senza uso di tunnel, ma presuppone che ogni chiamata `connect` a un server OTIP sia preceduta da una richiesta DNS per mantenere aggiornato l'indirizzo (come nel caso di SSH). Questo causa difficoltà quando gli agenti non si comportino come previsto, come visto nel caso paradigmatico dei Web browser, che sono tipicamente poco prevedibili e differenti tra loro nei comportamenti. Non sarebbe da escludere l'implementazione di un tunnel OTIP anche per TCP, in modo da accertarsi che ogni singola connessione sia indirizzata correttamente. Purtroppo l'utilizzo di un tunnel implicherebbe ulteriore lavoro e configurazione dal lato del client, che sarebbe ideale poter rimuovere completamente al di fuori dell'uso di un risolutore DNS specializzato.

Come risultato ottenuto dal lavoro, VdeDNS può essere un primo strumento di base per permettere l'introduzione all'utilizzo di indirizzi hash IPv6 basati su FQDN o temporanei a chiunque sia interessato a farne uso. Il proxy DNS può essere usato in congiunzione con gli strumenti di configurazione per indirizzi hash basati su FQDN già preesistenti, finché questi richiedano semplicemente un risolutore DNS idoneo e il caching

di domini per risoluzioni inverse. In particolare, il deployment di reti con l'uso di indirizzi basati su FQDN potrebbe avere successo come paradigma per la configurazione di una rete casalinga di dispositivi IoT o per la configurazione di una rete virtuale di processi IoTh, oltre che per l'uso in sistemi di ampia scala come data center. Usato invece in congiunzione con i nuovi strumenti sviluppati per OTIP, è possibile iniziare a utilizzare sperimentalmente il metodo di protezione di servizi tramite IP variabili proposto, finché questi servizi siano compatibili con l'implementazione attuale (come ad esempio SSH e parzialmente i Web server). Anche in questo caso sia utenti privati sia altri enti potrebbero voler aggiungere un layer di protezione a qualche servizio privato potenzialmente vulnerabile. Se il riscontro dell'utilizzo sperimentale di OTIP dovesse essere positivo, non sarebbe da escludere che potrebbe avvenire una standardizzazione della gestione delle chiamate DNS all'interno di un programma, allo scopo di rendere un più ampio numero di servizi TCP naturalmente compatibile.

Sviluppi futuri

VdeDNS può essere ancora ampliato per supportare ulteriori feature utili, per quanto sia in possesso di tutte quelle indispensabili per l'attuale stato dell'arte dei proxy DNS.

La prima e principale feature mancante, che non è stata implementata principalmente per motivi di tempo, è il supporto per il protocollo Domain Name System over Transport Layer Security (DNS over TLS). [14] Questo protocollo utilizza TCP per trasmettere pacchetti DNS utilizzando un canale di trasmissione sicuro tramite TLS, comunicando sulla porta 853. Al di fuori dell'uso di socket TLS, che sono tipicamente implementati da librerie specializzate, le specifiche per il trasporto DNS sono esattamente le stesse di TCP: si dovrebbe poter quindi in realtà riutilizzare quasi nella sua interezza l'implementazione TCP già in uso su VdeDNS.

Altre feature implementabili su VdeDNS potrebbero essere un sistema di caching per migliorare ulteriormente la performance o strutture dati più intelligenti per la ricerca di domini e record locali. Al momento, come già accennato, VdeDNS non esegue alcun tipo di caching, se non per il caso speciale delle risoluzioni inverse di domini *HASH*. Potrebbe essere implementata banalmente una cache per le richieste generiche e *HASH* che necessitino di forwarding, ma le richieste *OTIP* necessiterebbero un sistema più complesso, richiedendo tempi di caching variabili in base al timeslice corrispondente. Le strutture dati di domini e record locali sono al momento implementate come una lista non ordinata, quindi sarebbe possibile ottimizzarle e velocizzare la ricerca di elementi.

Oltre agli strumenti relativi all'uso di OTIP, si sarebbe potuto aggiornare anche l'assegnatore di indirizzi *HASH*. Come visto in precedenza, sono state proposte principalmente due opzioni per gestire l'assegnamento di indirizzi basati su FQDN:

DHCP Configurazione stateful

SLAAC Configurazione stateless

La prima opzione è quella già sviluppata e contenuta in hashdns [26], un server DHCP che quando riceve query che includano un FQDN esegue una richiesta a un DNS specializzato per poter restituire il corretto dominio *HASH* all'host. La seconda opzione, non ancora sviluppata, sarebbe un demone che permetta a un host di autoconfigurarsi in base al suo FQDN utilizzando un indirizzo base ottenuto da un router advertisement. Il problema

della seconda opzione è che, a differenza della prima, dovrebbe reimplementare al proprio interno la computazione dell'indirizzo *HASH*, poiché non avendo ancora un indirizzo IPv6 valido non potrebbe eseguire una query al DNS specializzato. Infatti, se la computazione non è centralizzata su un solo programma, c'è il rischio che demone e DNS possano implementare due differenti algoritmi, ad esempio nel caso di aggiornamenti.

Una terza alternativa per consentire l'uso di SLAAC e risolutore DNS potrebbe essere aprire un DNS specializzato sull'interfaccia di loopback (::1) o su un indirizzo link local (fe80::), in modo che il demone di configurazione non debba nuovamente implementare l'algoritmo di computazione di indirizzi e possa invece fare semplicemente una richiesta in locale.

Con questo lavoro si sono estese le fondamenta della ricerca sull'adozione di metodi di indirizzamento IPv6 basati su funzioni hash. Si è riusciti a mostrare la generale realizzabilità delle idee proposte attraverso esperimenti concreti, nonostante si sia incorsi in qualche limitazione causata dal comportamento di programmi esterni, fuori dal controllo dell'implementazione. In conclusione, si auspica che con la ricerca svolta possa nascere interesse nell'ulteriore sviluppo e nell'adozione delle tecnologie presentate, essendosi queste rivelate strumenti efficaci nel contesto odierno di system deployment e sicurezza.

Appendices

Appendice A

OTIP Web Server Tutorial

Let's setup a OTIP service using the reverse proxy and then either the `addrgen` executable or `vdedns`. We'll have this service on a fc00:aaaa::/64 network with a **mydomain.otip** fully qualified domain name.

A.1 Proxy Setup

First of all we are going to set a tap interface up and plug it to a VDE network for the proxy to be able to be reachable from the outside.

```
sudo ip tuntap add mode tap tap0;
sudo ip addr add 10.0.0.254/24 dev tap0;
sudo ip addr add fc00:aaaa::42/64 dev tap0;
sudo ip link set tap0 up;
```

We are going to use a standard VDE socket so:

```
vde_plug tap://tap0 switch:///tmp/vde
```

A.1.1 Proxy on kernel stack

Easiest setup for the proxy is to be bound to any interface of the kernel networking stack.

- The server the proxy is going to serve will be bound to loopback interface to prevent outside access
- The virtual networking stacks created by the proxy are going to use our tap0 interface as a gateway
- We are going to choose port 8888 as the server port (the one the proxy will connect to) and 8080 as the proxy port (the one the clients will connect to)

Our `test.otip` proxy configuration file is going to look like:

```
#otip server domain name
fqdn = mydomain.otip

#target server address
```

```
addr = 127.0.0.1

#target server port
port = 8888

#otip stack base address
baseaddr = fc000:aaaa::

#vde vnl
vde:///tmp/vde

#otip stack gateway
gw = 10.0.0.254

#otip info
pswd = otipsecret
timeslice = 32
offset = 8
```

Let's now run the proxy with:

```
./proxy 8080 test.otip
```

At this point the proxy will start printing its current address. You should be always able to ping the latest one using `ping6`.

A.2 Enabled server setup

Easiest example to setup is probably a web server. Open up a simple web server on a directory of choice using python:

```
python3 -m http.server -b 127.0.0.1 8888
```

We can now try out a simple connection using `curl` and `addrgen`.

```
curl [$(./addrgen mydomain.otip otipsecret fc00:aaaa::)]:8080
```

This should respond serving your web server main page.

A.3 Access using vdedns

Add an OTIP object rule to vdedns configuration:

```
{
    dom=mydomain.otip;
    pswd=otipsecret;
    time=32;
}
```

And a new local record (unless you have your own public domain with an IPv6 address):

```
{
    dom=mydomain.otip;
    ip6=("fc00:aaaa::");
}
```

A minimum `otiptest.cfg` sample file is below:

```
dns_servers = (
    "80.80.80.80"
);

rules = {
    otip = (
        {
            dom="mydomain.otip";
            pswd="otipsecret";
            time=32;
        }
    );
};

records = (
    {
        dom="mydomain.otip";
        ip6=("fc00:aaaa::");
    }
);
```

Now run:

```
sudo vdedns -c otiptest.cfg
```

And then change your local `/etc/resolv.conf` file adding on top:

```
nameserver 127.0.0.1
```

You can now use `curl mydomain.otip:8080` or access directly from your browser with **http://mydomain.otip:8080**.

Bibliografia

[1] Ray Bellis. *DNS Proxy Implementation Guidelines*. Rapp. tecn. 5625. RFC Editor, ago. 2009.

[2] Sebastian Castro et al. «Understanding and Preparing for DNS Evolution». In: vol. 6003. Mar. 2010. ISBN: 978-3-642-12364-1. DOI: 10.1007/978-3-642-12365-8_1.

[3] J. Damas, M. Graff e P. Vixie. *Extension Mechanisms for DNS (EDNS(0))*. Rapp. tecn. 6891. RFC Editor, apr. 2013.

[4] Renzo Davoli. «Internet of Threads». In: *Proc. of ICIW 2013 : The Eighth International Conference on Internet and Web Applications and Services* IARIA (International Academy, Research and Industry Association) (dic. 2014), pp. 100–105.

[5] Renzo Davoli. «IPv6 Hash-Based Addresses for Simple Network Deployment». In: *Proc. of AFIN 2013 : The Fifth International Conference on Advances in Future Internet* IARIA (International Academy, Research and Industry Association) (ago. 2013), pp. 15–20.

[6] Renzo Davoli. «OTIP: One Time IP Address». In: *proc. of ICSNC 2013: The Eighth International Conference on Systems and Networks Communications* (nov. 2013), pp. 154–158.

[7] Renzo Davoli. «VDE: virtual distributed Ethernet». In: *First International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities*. 2005, pp. 213–220.

[8] Renzo Davoli e Michael Goldweber. «msocket: Multiple stack support for the Berkeley socket API». In: (mar. 2012). DOI: 10.1145/2245276.2245390.

[9] Steve Deering e Bob Hinden. *IP Version 6 Addressing Architecture*. Rapp. tecn. 4291. RFC Editor, feb. 2006.

[10] J. Dickinson et al. *DNS Transport over TCP - Implementation Requirements*. Rapp. tecn. 7766. RFC Editor, mar. 2016.

[11] Louay Gammo et al. «Comparing and Evaluating epoll, select, and poll Event Mechanisms». In: *Proceedings of the 6th Annual Ottawa Linux Symposium* (gen. 2004).

[12] Fernando Gont. *A Method for Generating Semantically Opaque Interface Identifiers with IPv6 Stateless Address Autoconfiguration (SLAAC)*. Rapp. tecn. 7217. RFC Editor, apr. 2014.

[13] *Google IPv6 Statistics*. https://www.google.com/intl/en/ipv6/statistics.html. Accessed: 2021-04-20.

[14] Z. Hu et al. *Specification for DNS over Transport Layer Security (TLS)*. Rapp. tecn. 7858. RFC Editor, mag. 2016.

[15] Mwawi Nyirenda Kayuni et al. «Generating Unlinkable IPv6 Addresses.» In: *SSR*. A cura di Liqun Chen e Shin'ichiro Matsuo. Vol. 9497. Lecture Notes in Computer Science. Springer, 2015, pp. 185–199. ISBN: 978-3-319-27151-4. URL: http://dblp.uni-trier.de/db/conf/secsr/ssr2015.html#KayuniKLMY15.

[16] Paul Mockapetris. *Domain names - implementation and specification*. Rapp. tecn. 1035. RFC Editor, nov. 1987.

[17] Thomas Narten, Richard Draves e Suresh Krishnan. *Privacy Extensions for Stateless Address Autoconfiguration in IPv6*. Rapp. tecn. 4941. RFC Editor, set. 2007.

[18] Thomas Narten, Tatsuya Jinmei e Susan Thomson. *IPv6 Stateless Address Autoconfiguration*. Rapp. tecn. 4862. RFC Editor, set. 2007.

[19] Rosalea Robers, Geoff Huston e Thomas Narten. *IPv6 Address Assignment to End Sites*. Rapp. tecn. 6177. RFC Editor, mar. 2011.

[20] W. Stallings. «IPv6: the new Internet protocol». In: *IEEE Communications Magazine* 34.7 (1996), pp. 96–108. DOI: 10.1109/35.526895.

[21] W. Richard Stevens. *UNIX Network Programming*. USA: Prentice-Hall, Inc., 1990. ISBN: 0139498761.

[22] VirtualSquare Team. *libioth*. https://github.com/virtualsquare/libioth. Nov. 2020.

[23] VirtualSquare Team. *libvdestack*. https://github.com/rd235/libvdestack. Dic. 2016.

[24] VirtualSquare Team. *lwipv6*. https://github.com/virtualsquare/view-os/tree/master/lwipv6. Feb. 2014.

[25] VirtualSquare Team. *picoxnet*. https://github.com/virtualsquare/picoxnet. Gen. 2020.

[26] VirtualSquare Team. *vde_dnsutils*. https://github.com/rd235/vde_dnsutils. Gen. 2017.

[27] VirtualSquare Team. *vdens*. https://github.com/rd235/vdens. Dic. 2016.

[28] VirtualSquare Team. *vdeplug4*. https://github.com/rd235/vdeplug4. Set. 2016.

[29] S. Ziegler e L. Ladid. «Towards a Global IPv6 Addressing Model for the Internet of Things». In: *2016 30th International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. 2016, pp. 622–627. DOI: 10.1109/WAINA.2016.178.

[30] S. Ziegler et al. «Evaluation and recommendations on IPv6 for the Internet of Things». In: *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*. 2015, pp. 548–552. DOI: 10.1109/WF-IoT.2015.7389113.