UNIVERSITÀ DI PISA

Dipartimento di Informatica

Corso di Laurea Triennale in Informatica

Implementazione di un network overlay distribuito per comunicazioni sicure

Relatore:

Luca Deri

Candidato:

Francesco Carli

Anno Accademico 2019/2020

Indice

1 Introduzione **3**

1.1 Struttura della tesi 5

2 Obiettivi e Stato dell'Arte **6**

2.1 Obiettivi 6

2.2 Stato dell'Arte: Peer-to-Peer 10

2.3 Stato dell'Arte: Distributed Hash Tables 14

2.3.1 Kademlia 18

2.3.2 Chord 20

2.3.3 Koorde 22

2.4 Stato dell'Arte: Software VPN 24

2.4.1 Tailscale 24

2.4.2 ZeroTier 26

2.4.3 FreePN 28

3 Architettura del Sistema **29**

3.1 Supernode 32

3.2 Edge 34

3.3 Crittografia in n2n 36

4 Implementazione del Software **40**

4.1 Software: Supernode 42

4.1.1 Federazione di supernodi 44

4.1.2 REGISTER_SUPER_ACK Payload 47
4.1.3 Processo di re-registrazione ed eliminazione 50
4.2 Software: Edge 51
4.2.1 Strategia di selezione basata su RTT 53
4.2.2 Strategia di selezione basata su carico 56
4.2.3 Confronto delle strategie 59

5 Validazione **63**
5.1 Attacchi esterni 63
5.2 Attacchi interni 70
5.2.1 Indirizzi MAC duplicati 70
5.2.2 Indirizzi MAC falsificati 73
5.3 Risultati 75

6 Conclusione **76**

7 Lavori futuri **78**

8 Appendice **80**
8.1 Codice sorgente 80

Riferimenti **81**

Capitolo 1

Introduzione

Il progetto di questo studio tratta l'estensione delle comunicazioni del software n2n (network to network). Esso è un software open source che consente di creare dei network overlay sicuri, ossia delle reti di calcolatori logiche o fisiche poste al di sopra di altre reti. Dato che ormai i sistemi IoT (Internet of Things) sono ampiamente diffusi e data la futura disponibilità delle reti 5G, si avranno in futuro reti sempre più potenti e più grandi in termini di dispositivi connessi. Essi dovranno operare in reti virtuali sicure in modo che i meccanismi di sicurezza di cui sono dotati, spesso molto semplici, consentano loro di funzionare correttamente. Questo è il motivo per cui si utilizzano i network overlay, andando a isolare queste risorse e rendendo sicure le loro comunicazioni.

Gli elementi principali di n2n sono i supernodi e i nodi edge. Un supernodo consente ai nodi edge di annunciarsi e scoprire altri nodi della rete. Un nodo edge è invece un nodo che farà parte della rete virtuale. In n2n una rete virtuale condivisa tra più nodi è chiamata comunità, un singolo supernodo può supportare più comunità ed un singolo computer potrà far parte di più comunità in contemporanea. I membri di una comunità potranno anche utilizzare una chiave di cifratura per cifrare i messaggi scambiati tra loro. Inizialmente n2n tenta di stabilire, se possibile, una connessione diretta peer-to-peer tra i nodi edge utilizzando il protocollo UDP e, se ciò non risulta possibile, sarà il supernodo ad occuparsi anche dell'inoltro dei pacchetti.

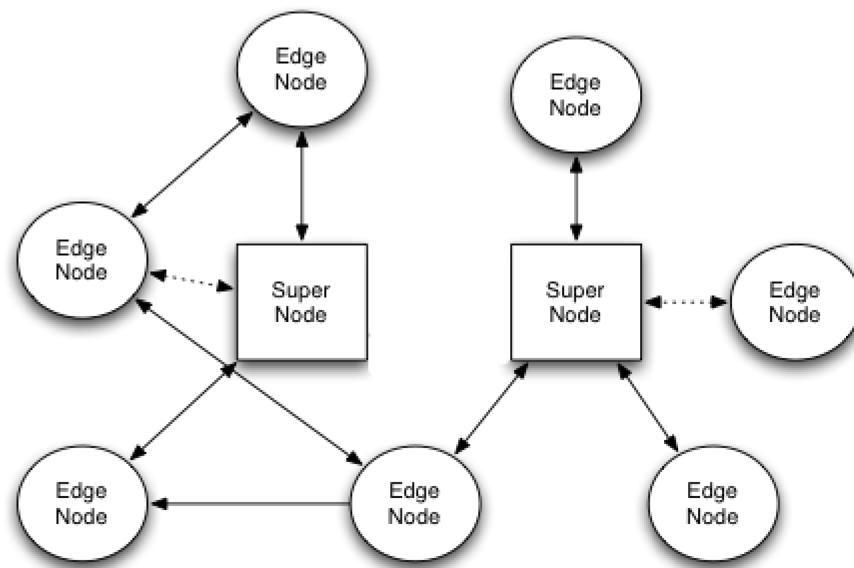

Figura 1: Architettura di n2n.

Il problema a seguito del quale si è sviluppato questo lavoro è dato dal fatto che il paradigma di comunicazione di n2n è semi-centralizzato (Figura 1). Ciò lo rende debole e instabile su dispositivi critici (es. sensori IoT) e in reti con comunicazioni intermittenti. Infatti, con il paradigma di comunicazione descritto in precedenza, la comunicazione tra nodi edge di una comunità è gestita da un supernodo, il quale potrebbe improvvisamente smettere di funzionare. Ogni supernodo in n2n è dotato di un supernodo di back-up, ma se ad esempio anch'esso divenisse irraggiungibile, si perderebbero tutte le informazioni di quel supernodo e le comunità da esso gestite non sarebbero più in grado di comunicare. L'obiettivo di questo progetto è quindi quello di estendere il protocollo di comunicazione di n2n rendendolo veramente distribuito, utilizzando una Distributed Hash Table (DHT) come struttura dati per tenere traccia dei nodi e delle relative configurazioni di rete.

1.1 Struttura della tesi

Nel Capitolo 2 verrà definito l'obiettivo di questo progetto. In più, si approfondiranno i vari argomenti affrontati e si mostreranno i software attualmente disponibili per quanto riguarda le reti virtuali sicure.

Nel Capitolo 3 si illustrerà in modo più approfondito il comportamento di n2n, le entità che fanno parte di una sessione tipica e la modalità con cui n2n rende una comunicazione sicura.

Nel Capitolo 4 saranno mostrate le scelte implementative attuate al fine di raggiungere gli obiettivi preposti, motivando i perché di tali scelte.

Il Capitolo 5 è dedicato alla fase di testing e ai risultati ottenuti, valutando le prestazioni del software sotto opportune condizioni.

Nei capitoli finali sono contenute le conclusioni del progetto, le possibili migliorie sviluppabili in futuro, l'appendice con il codice sorgente del software n2n ed i riferimenti.

Capitolo 2

Obiettivi e Stato dell'Arte

In questo capitolo sono inizialmente forniti i motivi per i quali è stato deciso intraprendere questo progetto di tirocinio e gli obiettivi da raggiungere.
Nel seguito, sono illustrati in maniera più approfondita e tecnica gli argomenti necessari alla comprensione del software n2n e del suo funzionamento, vale a dire tecnologia peer-to-peer e distributed hash tables (DHTs). Infine, sono stati riportati tre esempi di software VPN esistenti per evidenziare le loro caratteristiche e sottolineare le differenze con n2n, in particolare riguardo il protocollo di comunicazione adottato.

2.1 Obiettivi

Come già introdotto nel Capitolo 1, l'obiettivo di questo progetto di tirocinio è quello di ridefinire il protocollo di comunicazione esistente del software n2n, al fine di renderlo totalmente distribuito. Il lavoro si è basato quindi inizialmente su un'analisi approfondita del codice esistente, per comprenderne a pieno il funzionamento del software. Dopodiché, data la sua natura open source, è avvenuto anche un confronto con vari sviluppatori di n2n per definire quali sarebbero state le migliori funzionalità da introdurre per raggiungere l'obiettivo prefissato. Essendo un prodotto già in funzione da molti anni e utilizzato anche da persone o enti per motivi lavorativi, è stato molto importante definire con chiarezza gli obiettivi da raggiungere e le modifiche da implementare, per evitare di realizzare funzionalità che alterassero il corretto funzionamento del software.

Il problema principale a seguito del quale si è intrapreso questo progetto è legato alla necessità di avere reti virtuali completamente sicure per gestire l'enorme quantità di dispositivi IoT [5] presenti nelle reti odierne e, in un futuro ormai vicino, l'avvento delle reti 5G [39].
L'obiettivo posto è stato dettato dal fatto che, analizzando i vari software VPN disponibili, tra cui quelli illustrati nel seguito del capitolo come Tailscale [27], ZeroTier [38] e FreePN [12], è stato possibile notare che essi si basano su un'architettura ed un protocollo client-server o comunque semi-centralizzato. Ciò sta a significare che in reti instabili e/o in presenza di devices potenzialmente critiche si potrebbero verificare delle interruzioni di servizio, attacchi di sicurezza, irraggiungibilità di alcuni membri della rete o, più in generale, problemi di affidabilità, robustezza e resilienza della rete stessa.

Le reti di comunicazione wireless sono spesso soggette a minacce alla sicurezza. Per quanto riguarda i sistemi basati su IoT, essi sono generalmente complessi a causa dell'enorme impatto su tutti gli aspetti della vita umana (ad esempio sicurezza, salute, mobilità, efficienza energetica, sostenibilità ambientale, ecc.) e delle tecnologie impiegate per consentire lo scambio di dati tra dispositivi [5, 23]. L'analisi di studi e documenti recenti mostra che la maggior parte dei problemi sorgono a causa di un numero crescente di dispositivi collegati che causano un aumento della domanda di traffico. Altre questioni sono invece legate all'integrazione di varie tecnologie, ambienti eterogenei, maggiore archiviazione dei dati e richieste di elaborazione, rischi per la privacy e la sicurezza, ecc. A causa del sempre maggiore utilizzo dei dispositivi IoT, le reti sono soggette a vari attacchi alla sicurezza, ecco perché la distribuzione di protocolli di sicurezza e privacy efficienti nelle reti IoT diventa fondamentale per garantire riservatezza, autenticazione, controllo degli accessi, integrità, ma anche disponibilità e mobilità.

La disponibilità dei servizi [5] è uno dei temi fondamentali da affrontare per gestire adeguatamente le dinamiche dei sistemi IoT o più in generale di qualsiasi sistema

più o meno instabile. Disponibilità significa che un servizio o un'applicazione dovrebbero essere accessibili ovunque e in qualsiasi momento per ogni entità autorizzata. Tutto questo richiede meccanismi per l'interoperabilità, la rilevazione e il ripristino in caso di alcune operazioni non previste. È necessario implementare un sistema di monitoraggio appropriato, protocolli e meccanismi di recupero per consentire la robustezza del sistema. Infatti, le comunicazioni in sistemi instabili, quali i sistemi IoT, soffrono di disponibilità intermittente che può causare interruzione del servizio e perdita delle configurazioni della rete.

La mobilità è un altro grande problema dei sistemi odierni poiché i servizi sono forniti ad utenti mobili. I dispositivi IoT, ma anche smartphone, laptop, tablet, ecc. sono in grado di spostarsi, entrando ed uscendo dalla rete, rendendo possibile il verificarsi di frequenti modifiche alla topologia della rete stessa. L'obiettivo è quello di creare un sistema robusto nonostante questi cambiamenti dinamici. Inoltre, alcune implementazioni di sistemi implicano che i dispositivi debbano conoscere la loro posizione e l'ambiente circostante.

Per cui l'obiettivo di questo progetto è quello di risolvere le varie problematiche analizzate in precedenza andando a modificare il protocollo di comunicazione ed in parte l'architettura del software n2n. Essendo un software semi-centralizzato, che fa molto affidamento sulla struttura del supernodo, è facile comprendere che un attacco mirato verso questa struttura o un'improvviso malfunzionamento di essa potrebbero avere conseguenze pesanti sugli utenti della rete, in quanto essi potrebbero non essere più in grado di comunicare tra loro e le loro configurazioni di rete potrebbero andare perse.

Obiettivi
Comunicazione tra supernodi
Propagazione stato della rete
Strategia di selezione su nodi edge
Resistenza ad attacchi interni

Nel seguito di questo capitolo si approfondiranno gli argomenti alla base di una rete virtuale sicura distribuita, ossia la tecnologia peer-to-peer (P2P) [11, 17, 25] e le Distributed Hash Tables (DHTs) [6, 9, 31]. Infine, saranno presentati tre diversi software per la realizzazione di reti virtuali sicure, illustrando il loro funzionamento e le loro caratteristiche principali.

2.2 Stato dell'Arte: Peer-to-Peer

Il termine peer-to-peer (P2P) indica un tipo di architettura logica di una rete informatica in cui i nodi non sono gerarchizzati secondo il paradigma client-server, ma sono paritari ed agiscono sia da client che da server. Questo paradigma è nato inizialmente come servizio di condivisione di file musicali, immagini, ecc. Una rete P2P è quindi un'architettura di calcolo distribuita in cui i membri condividono tra loro le proprie risorse (es. RAM, potenza di calcolo, banda, ecc...) in modo da sopperire alla mancanza di un'entità centrale di controllo.

La maggiore differenza tra una rete P2P e un'architettura client-server sta nella decentralizzazione/centralizzazione delle risorse e delle informazioni.
Nel caso della rete P2P tutti i nodi parteciperanno in maniera uguale alla gestione e al mantenimento della rete stessa: il funzionamento, dunque, non dipenderà da un unico nodo centrale e la topologia della rete potrà essere facilmente rivista per adattarsi a una situazione in continua evoluzione.
Nell'architettura client-server, invece, il carico della gestione dell'infrastruttura e delle informazioni è spostato verso un unico nodo centrale (il server), cui i vari nodi fanno riferimento per cercare e consultare le risorse desiderate. Questo approccio rende questo tipo di rete completamente dipendente dalla presenza del server centrale: se questo, per qualche ragione, smettesse di funzionare l'intera rete rimarrebbe bloccata.

I principali vantaggi delle reti P2P rispetto alle reti centralizzate sono dovuti: alla gestione più semplice della rete, all'efficacia nella condivisione delle risorse, alla trasmissione dati più veloce, alla scalabilità e all'ottima capacità di rigenerazione. I vantaggi saranno dunque proporzionali al numero di membri di una rete paritaria: più nodi ci sono, più la rete in questione avrà potenza e velocità di calcolo paragonabile ad un potente server. Tutti i nodi hanno quindi la stessa importanza, non esiste un controllo centralizzato, i sistemi sono distribuiti ed i nodi sono dinamici, cioè possono uscire ed entrare nella rete P2P in ogni momento.

Non essendoci un ente centrale che mette in comunicazione i vari componenti della rete, nel paradigma P2P ci si avvale solitamente di una overlay network [11], ossia una rete sovrapposta virtuale tramite la quale si va a rappresentare una sotto-rete che sta al di sopra della rete fisica vera e propria e che serve soprattutto per mappare i nodi e formare una topologia della rete stessa. Esistono due tipologie di reti P2P in base ai collegamenti tra i nodi e alla loro distribuzione [11, 17]:

- *Reti non strutturate*: come suggerisce anche il nome, una rete non strutturata è caratterizzata da un'apparente disorganizzazione ed è formata da nodi che creano collegamenti casuali con altri nodi della rete. Si tratta, dunque, di reti di facile formazione, che non richiedono il rispetto di parametri particolarmente stringenti. Allo stesso tempo, però, la mancanza di una struttura e di un'organizzazione interna rende particolarmente complessa e lunga la ricerca di file o risorse all'interno della rete: la richiesta, infatti, dovrà essere inviata a tutti i nodi che condividono il file. Ciò, ovviamente, genera un gran volume di traffico, senza la certezza di riuscire a individuare la risorse cercata.

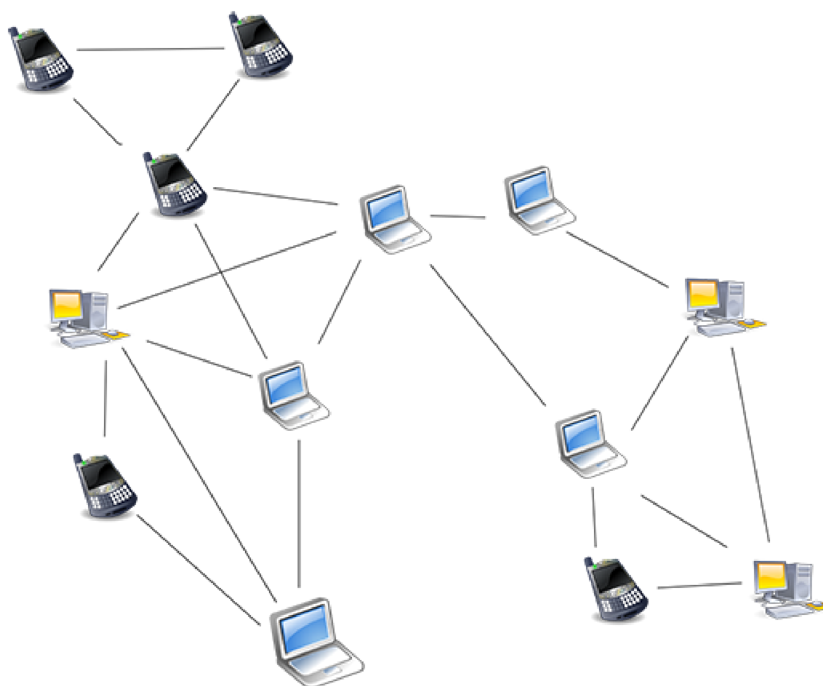

Figura 2: Esempio di rete P2P non strutturata.

- *Reti strutturate*: sono caratterizzate da una topologia specifica, la quale assicura che ogni nodo possa efficientemente cercare e trovare una risorsa, o un altro nodo, anche se essa dovesse essere particolarmente rara. Per facilitare questo compito, ogni rete strutturata integra una DHT, all'interno della quale a ogni risorsa corrisponde un codice identificativo univoco. Per far sì che il flusso del traffico non incontri ostacoli, è necessario che ogni nodo conservi un elenco dei nodi "vicini", che rispetti criteri molto vincolanti. Questo può costituire un grosso limite in termini di efficienza, nel caso in cui la rete sia caratterizzata da nodi che entrano o escono dalla rete molto frequentemente.

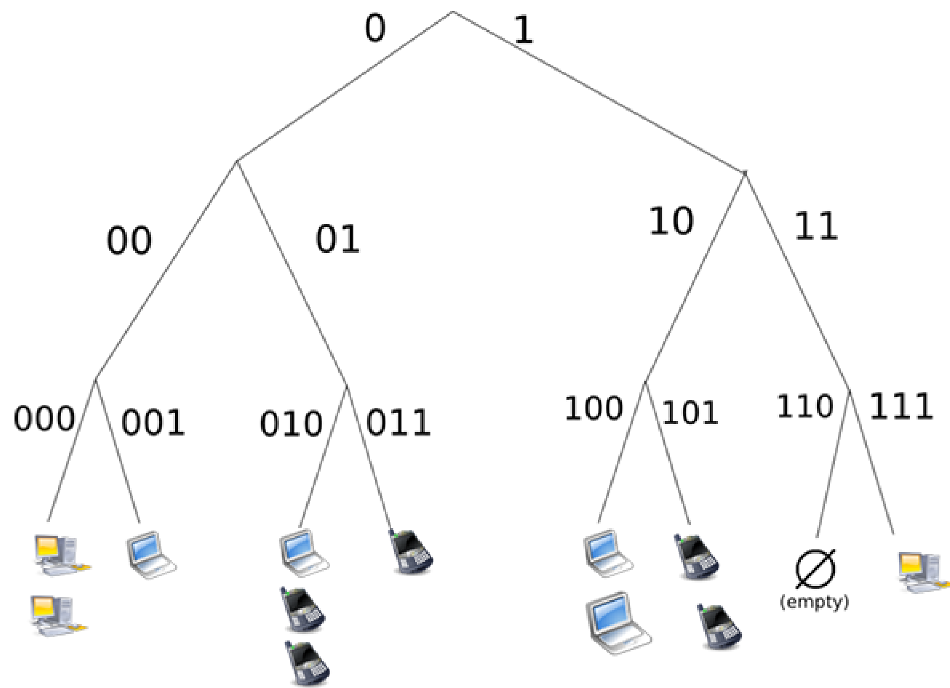

Figura 3: Esempio di rete P2P strutturata.

Un'architettura di rete P2P è caratterizzata da un'elevata scalabilità e capacità di "autorigenerazione". Nel caso in cui un nodo smetta di funzionare – o, semplicemente, si disconnetta – la rete continuerà a funzionare senza grossi problemi: lo scambio di dati o la ricerca di risorse proseguirà lungo strade alternative rispetto a quella che transitava attraverso il nodo non più disponibile. Si aggira, in questo modo, il collo di bottiglia che, in alcuni casi, è rappresentato dall'affidarsi al funzionamento di un

unico server (o un'unica struttura centralizzata). La distribuzione delle risorse, inoltre, permette di raggiungere velocità di download molto elevate: potendo scaricare la stessa risorsa da più fonti contemporaneamente, sarà possibile sommare la banda garantita da ogni singolo nodo.

L'elevata velocità media rende le reti P2P particolarmente adatte alla condivisione e al download dei file. Protocolli di comunicazione ”paritari”, dunque, sono utilizzati sia nell'ambito del content delivery sia nel file sharing (spesso sinonimo del termine ”scaricare gratis”). Molti servizi di condivisione di contenuti ”legali” (come ad esempio Spotify) adottano una soluzione ibrida, affiancando una struttura client-server a una peer-to-peer: ciò consente al servizio di bilanciare il carico di lavoro tra la propria infrastruttura (server e dorsali di comunicazione) e le risorse informatiche messe a disposizione dai singoli utenti.

2.3 Stato dell'Arte: Distributed Hash Tables

Le Distributed Hash Tables (DHTs), in italiano tabelle hash distribuite, sono strutture dati decentralizzate che memorizzano coppie chiave-valore e consentono operazioni come una qualsiasi tabella hash [9]. Si tratta quindi di una sorta di database distribuito in grado di archiviare e recuperare le informazioni associate a una chiave in una rete di nodi che possono entrare e uscire da essa in qualsiasi momento. I nodi si coordinano tra di loro per bilanciare e memorizzare i dati nella rete senza alcuna parte centrale di coordinamento. La capacità di distribuire i dati tra i nodi è in forte contrasto con il modello blockchain in cui, ad esempio, ogni nodo ha una copia dell'intero file e non solo una porzione di esso.

Il principale vantaggio di una DHT è dovuto al fatto che i nodi possono essere aggiunti o rimossi minimizzando il lavoro di riorganizzazione delle chiavi. Infatti, in una tabella hash classica ogni aggiunta o rimozione di un elemento dalla struttura richiede la ridistribuzione di tutte le chiavi, mentre nelle DHTs ciò non avviene poiché non ci sarà un singolo nodo che avrà tutto il contenuto della DHT. Ogni nodo di una DHT è responsabile di una o più chiavi del keyspace e potrà restituire in modo efficiente le informazioni associate ad ognuna di esse. Le DHT sono utilizzate per gestire un vasto numero di nodi, con ingressi continui nella rete o guasti improvvisi, e per implementare servizi più complessi come file systems distribuiti, sistemi P2P di file sharing, ecc.

Le caratteristiche di una DHT sono principalmente:

- *Decentralizzazione*: i nodi non hanno nessun ente di coordinamento centrale.
- *Scalabilità*: il sistema funziona anche con grandi quantità di nodi.
- *Fault Tolerance*: il sistema funziona anche con nodi che entrano ed escono continuamente dalla rete o che sono soggetti a guasti e malfunzionamenti frequenti.

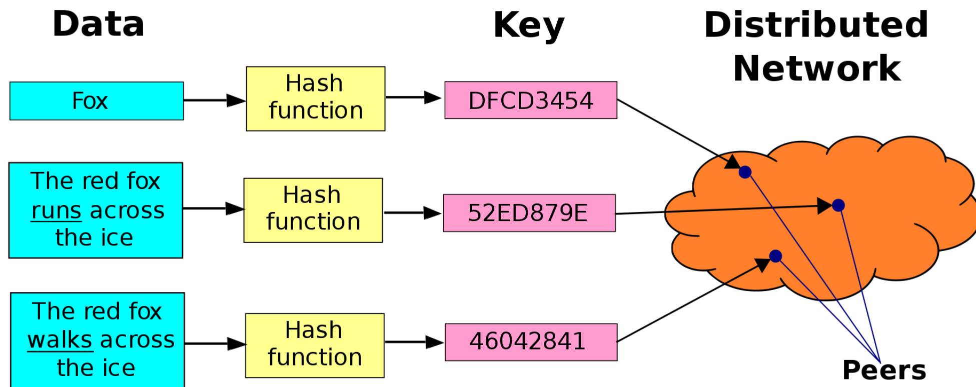

Figura 4: Esempio di una DHT e di come il suo contenuto viene suddiviso tra i nodi della rete.

Una DHT è solitamente costruita attorno ad un keyspace (ad esempio set di stringhe di 160 bit) e l'appartenenza ad esso è diviso tra i nodi partecipanti in base ad uno schema ben definito. Il tipico funzionamento di una DHT per memorizzare e poi recuperare un dato è il seguente [31]: sia il keyspace un set di stringhe e sia dato un file caratterizzato da nome e data, si calcola l'hash del nome del file, generando una chiave k, che può essere utilizzata per inviare un'informazione ad un nodo della DHT. Il messaggio sarà quindi inoltrato di nodo in nodo attraverso l'overlay network fintanto che non raggiungerà il nodo responsabile per la chiave k, in accordo alle regole per il partizionamento del keyspace. Una volta raggiunto il nodo responsabile della chiave k, qui saranno memorizzate le informazioni (data) sul file. Successivamente, qualsiasi altro client potrà recuperare le informazioni di quel particolare file, calcolando l'hash di quel particolare documento per ottenere la chiave k, e chiedere ad ogni nodo della DHT le informazioni a lui associate. La richiesta verrà inoltrata al nodo responsabile per la chiave k, il quale risponderà con una copia delle informazioni memorizzate.

Poiché i nodi DHT non memorizzano tutti i dati, è necessario un livello di instradamento in modo che qualsiasi nodo possa individuare il nodo responsabile di una particolare chiave [8]. Il meccanismo di funzionamento della tabella di instradamento e il modo in cui essa viene aggiornata quando i nodi si uniscono e lasciano la rete è un elemento chiave di differenziazione tra i diversi algoritmi DHT esistenti. In generale,

i nodi DHT memorizzeranno solo una parte delle informazioni di instradamento nella rete. Ciò significa che quando un nodo si unisce alla rete, le sue informazioni devono essere diffuse solo a un numero limitato di nodi. Questo implica che ogni nodo contiene solo una parte della tabella di instradamento ed il processo di ricerca o archiviazione di una coppia chiave-valore richiederà il contatto di più nodi.

Uno dei modi più semplici per organizzare la disposizione dei nodi all'interno di una DHT è di utilizzare una overlay network con una rappresentazione circolare. In questo modo la rimozione o l'aggiunta di un nuovo nodo modifica solo il set di chiavi dei nodi adiacenti e non di tutta la struttura. Dal momento che la riorganizzazione di un set di chiavi di un nodo comporta un trasferimento di oggetti da quel nodo ad un altro della DHT, minimizzare questo fenomeno consente di trattare anche nodi più dinamici, che entrano ed escono di continuo. Ogni nodo mantiene un insieme di collegamenti agli altri nodi - in generale $O(logN)$, con N numero di nodi della DHT, per consentire delle riorganizzazioni rapide ed efficienti - i quali tutti insieme formano l'overlay network.

Tutte le varie implementazioni di DHT condividono una proprietà essenziale: per ogni chiave k, o un nodo possiede k oppure ha un link al nodo più vicino a k in termini di distanza dal keyspace [31]. A questo punto risulta facile inoltrare qualsiasi tipo di messaggio al nodo con chiave k utilizzando il seguente algoritmo greedy: ad ogni passo, si inoltra il messaggio al nodo il cui ID è il più vicino a k. Se non si trova nessun nodo, allora siamo giunti al nodo desiderato, che dovrà essere il proprietario della chiave k. Al di là della correttezza di questo tipo di strategia, vi sono due vincoli chiave: il numero massimo di passi successivi in un qualsiasi percorso (dilazione) deve essere basso, in modo tale che la richiesta sia soddisfatta velocemente, ed anche il numero massimo di vicini di ciascun nodo (il grado del nodo) deve essere basso, al fine di mantenere minimo o attenuare l'overhead di mantenimento.

Esistono diverse metriche per misurare le prestazioni degli algoritmi DHT e l'ottimizzazione di una generalmente tende a far peggiorare le altre. Queste metriche includono:

- *Grado*: numero di vicini con cui il nodo deve mantenere un contatto continuo.
- *Hop Count*: il numero di hop (salti) necessari per recapitare un messaggio da un nodo ad un altro.
- *Fault Tolerance*: grado di tolleranza ai guasti, quali nodi della rete potranno fallire senza influire sulle comunicazioni e sui dati.
- *Sovraccarico di manutenzione*: quanto spesso i messaggi vengono scambiati tra nodi vicini. Questo dato è alto quando siamo in una rete molto dinamica, in cui molti nodi entrano ed escono continuamente.
- *Bilanciamento del carico*: distribuzione il più possibile uniforme delle chiavi tra i nodi della rete.

Le caratteristiche delle DHTs ed il contesto in cui esse operano sono le cause principali per cui si è valutato il loro utilizzo come strutture dati ausiliarie per mantenere le varie configurazioni di rete dei nodi all'interno di n2n. Infatti, utilizzando le DHTs, si potrà gestire in maniera più efficiente i fault della rete, poiché si manterranno all'interno di una DHT, per un certo periodo di tempo, le configurazioni di quei nodi edge che hanno un comportamento altamente dinamico. In più, si potrà senza alcun problema affrontare anche situazioni più critiche. Se, ad esempio, un supernodo riuscisse a comunicare in qualche modo con gli altri supernodi della rete e periodicamente inviasse a loro e agli edge le configurazioni di rete conosciute, in poco tempo potremmo avere una panoramica generale di come la rete è strutturata in un dato istante. In questo scenario, eventuali guasti ad un supernodo non sarebbero più un problema, poiché i nodi edge verrebbero a conoscenza di altri supernodi della rete presso i quali registrarsi e del modo in cui raggiungerli. Per quanto riguarda le varie tipologie di algoritmi DHT esistenti, sono stati analizzati tre diversi algoritmi riportati nel seguito di questo capitolo.

2.3.1 Kademlia

Kademlia è un protocollo di rete P2P per DHTs che presenta una serie di funzionalità non offerte da nessun'altra DHT: infatti riduce al minimo il numero di messaggi di configurazione che i nodi devono inviare per conoscersi a vicenda [18]. Le informazioni sulla configurazione si diffondono automaticamente come effetto collaterale delle ricerche e nessun nodo ha abbastanza conoscenza e flessibilità per indirizzare una query attraverso percorsi a bassa latenza.
Ogni nodo è identificato da un numero (”ID nodo”), il quale non serve solo ai fini del suo riconoscimento, ma viene anche utilizzato dall'algoritmo Kademlia per localizzare valori (ad esempio l'immagine hash del nome di un file).
La rete definisce un concetto di distanza che permette di stabilire la prossimità tra due nodi cosicché, dato un nodo A è sempre possibile determinare se un nodo B risulta più vicino di un nodo C. Ogni nodo ha una conoscenza della rete che diminuisce all'aumentare della distanza da esso, e, quindi, ha un conoscenza molto dettagliata della rete nei propri dintorni, una discreta conoscenza della rete a media distanza e solo una conoscenza sparsa dei nodi molto lontani [33].
Per memorizzare un valore, il nodo richiedente calcolerà la chiave corrispondente (generalmente l'hash del valore da memorizzare), cercherà nella rete i nodi con ”ID nodo” più prossimi all'hash del valore da memorizzare e richiederà la memorizzazione del valore in tutti questi nodi. Quando si cerca una chiave, l'algoritmo esplora la rete in passi successivi e ad ogni passaggio ci si avvicina sempre più alla chiave cercata finché il nodo contattato non restituisce il valore oppure non ci sono più nodi da interrogare.

Molti dei vantaggi di Kademlia derivano dall'utilizzo della metrica XOR [32] per calcolare la distanza tra i nodi nello spazio delle chiavi. La distanza così calcolata non ha nulla a che vedere con la distanza geografica e due nodi che risultano vicini per l'algoritmo possono essere localizzati in due continenti differenti. L'algoritmo procede iterativamente nella ricerca delle chiavi attraverso la rete, avvicinandosi di un bit al risultato ad ogni passo compiuto. Ne consegue che una rete Kademlia con 2^n nodi,

richiederà al massimo n passi per trovare il nodo cercato. La tabella di indirizzamento di Kademlia consiste di una lista per ogni bit del "ID Nodo" (cioè se il "ID nodo" è lungo 128 bit, il nodo conserverà 128 di queste liste). Ogni elemento della lista contiene le informazioni necessarie a localizzare un nodo. Ogni lista corrisponde a una specifica distanza dal nodo. I nodi che finiscono nella ennesima lista devono avere almeno l'ennesimo bit differente da quello del nodo corrente. Nella letteratura di Kademlia la lista è chiamata k-bucket. Ogni k-bucket è quindi una lista contenente fino a k entrate che fanno riferimento a k nodi distanti k bit dal nodo considerato.

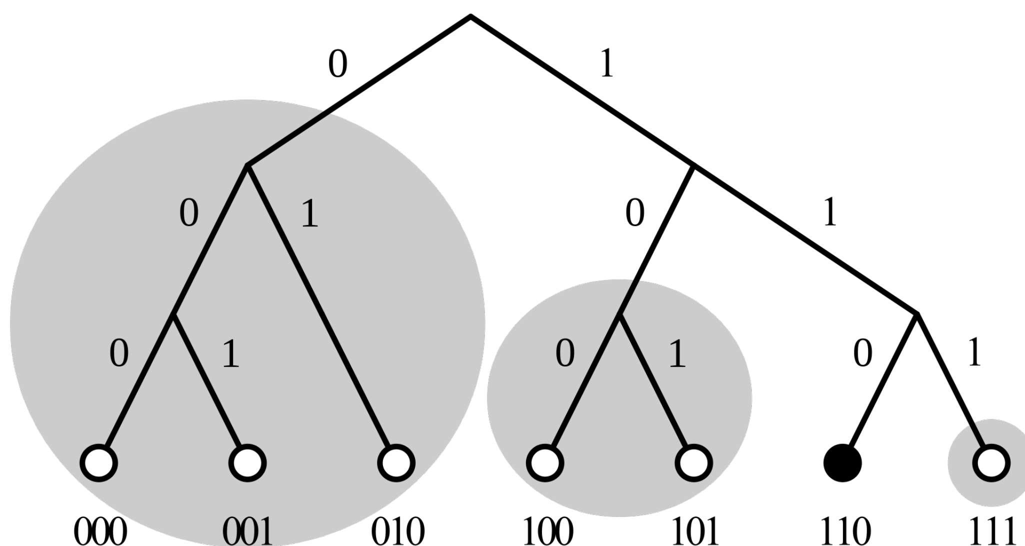

Figura 5: Con $n = 3$, ci sono otto nodi (2^3). Nell'esempio i nodi sono sette poiché uno non è considerato ed il nodo in esame è il sei (110). Ogni nodo della rete dispone di 3 k-bucket (cerchi grigi), uno per ciascun bit. I nodi 000, 001 e 010 sono candidati ad essere inclusi nel primo k-bucket; i nodi 100 e 101 sono candidati ad essere inclusi nel secondo k-bucket; il terzo k-bucket contiene solo il nodo 111.

2.3.2 Chord

Chord è un protocollo e un algoritmo P2P per DHTs che specifica come le chiavi vengano assegnate ai nodi e come un nodo possa scoprire il valore per una data chiave, individuando il responsabile di quest'ultima [29]. Il protocollo Chord supporta principalmente un'operazione: data una chiave, mappa la chiave su un nodo della rete. Chord utilizza il consistent hashing [28] per l'assegnazione delle chiavi ai nodi: esso tende a bilanciare il carico, poiché ogni nodo riceve all'incirca lo stesso numero di chiavi, e richiede un movimento relativamente piccolo delle chiavi quando i nodi entrano ed escono dalla rete. Il consistent hashing è uno schema di hashing distribuito che opera indipendentemente dal numero di oggetti in una tabella hash assegnando loro una posizione su un cerchio detto hash ring. Ciò consente a questi oggetti di scalare senza influire sul sistema complessivo.

Un nodo Chord richiede informazioni su altri nodi per un instradamento efficiente, ma le prestazioni si degradano regolarmente quando tali informazioni non sono aggiornate. Questo è importante nella pratica perché i nodi si uniranno e usciranno arbitrariamente dalla rete e la coerenza delle informazioni potrebbe essere difficile da mantenere.
L'utilizzo principale di Chord è quello di interrogare una chiave, ad esempio, per trovare il suo successore. L'approccio di base è passare la query al successore di un nodo, se non si riesce a trovare la chiave localmente. Questo porterà a un tempo di interrogazione $O(N)$ in cui N è il numero di nodi della rete.

Per evitare la ricerca lineare, Chord implementa un metodo di ricerca più veloce richiedendo a ciascun nodo di mantenere una finger table con al più m voci (con m numero di bit dell'hash della chiave k). La prima voce della tabella è l'immediato successore del nodo. Ogni volta che un nodo vuole cercare una chiave k, passerà la query al successore o predecessore più vicino a k nella sua finger table finché un nodo non scopre che la chiave è memorizzata nel suo immediato successore.

Con una finger table definita in questo modo, il numero di nodi che dovranno essere contattati per trovare un successore in una rete a N nodi è $O(logN)$.

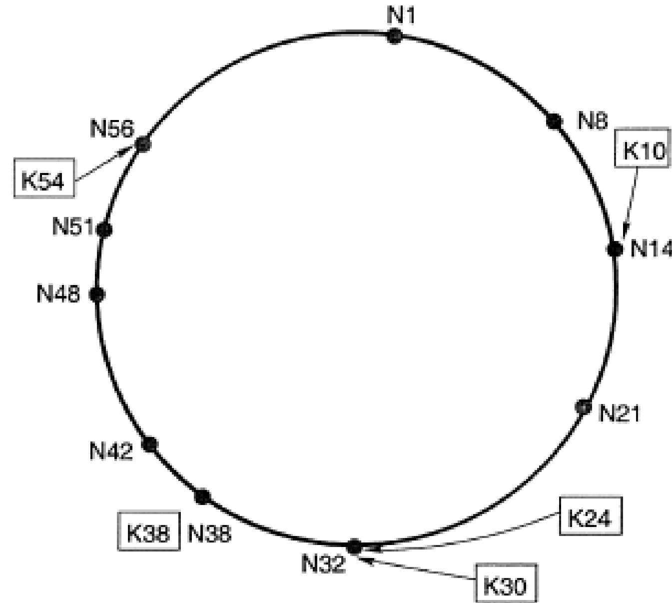

Figura 6: Rete di 10 nodi disposta ad anello (Chord ring) in cui sono memorizzate 5 chiavi. Per ogni chiave è mostrato il successore [26].

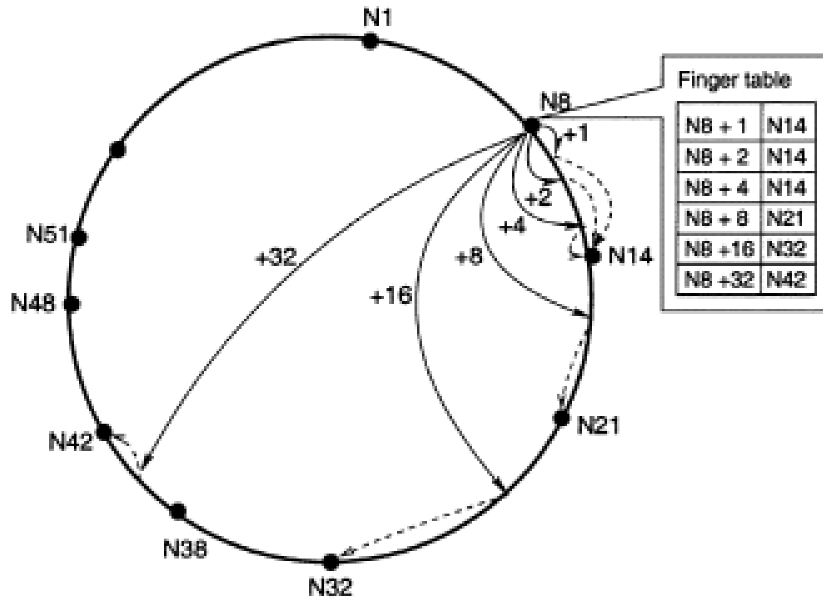

Figura 7: Contenuto della finger table del nodo N8. [26].

2.3.3 Koorde

Koorde è un protocollo per DHTs basato su Chord e sui grafici di de Bruijn [15, 34]. Ereditando la semplicità di Chord, Koorde impiega $O(logN)$ salti per nodo (dove N è il numero di nodi nella DHT) e $O(logN/log(logN))$ salti per richiesta di ricerca con $O(logN)$ vicini per nodo.

Come detto, Koorde basa il proprio funzionamento sul grafo di De Bruijn. In un grafo di questo tipo m-dimensionale, ci sono 2^m nodi, ognuno dei quali ha un ID univoco di m bit. Il nodo con ID i è connesso ai nodi 2i mod 2^m e 2i+1 mod 2^m. Grazie a questa proprietà, l'algoritmo di routing può instradare verso qualsiasi destinazione in m salti [15, 34].
L'instradamento di un messaggio dal nodo x al nodo y si ottiene prendendo il numero x in binario e spostando i bit di y uno alla volta finché il numero x non è stato sostituito da y. Ogni spostamento di bit corrisponde a un salto di instradamento al successivo indirizzo intermedio; l'hop è valido perché i vicini di ogni nodo sono i due possibili risultati dello spostamento di uno 0 o 1 sul proprio indirizzo. A causa della struttura dei grafici di de Bruijn, nel momento in cui l'ultimo bit sarà spostato, la query si troverà al nodo destinazione, il quale risponderà solamente se la chiave richiesta esiste.

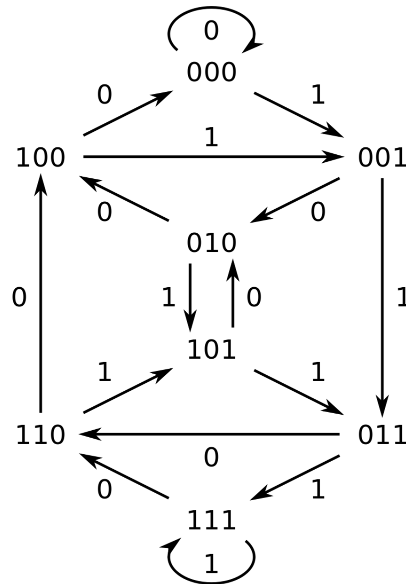

Figura 8: Grafo di De Bruijn con dimensione $m = 3$.

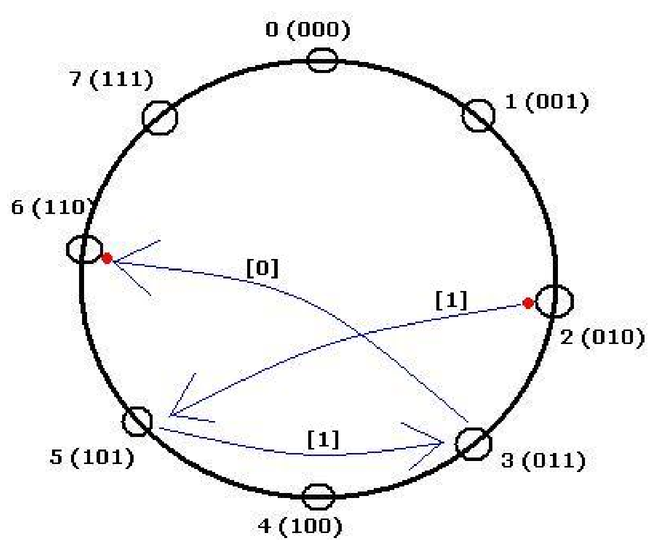

Figura 9: Esempio di come Koorde instrada il nodo 2 (010) verso il nodo 6 (110) avvalendosi di un grafo di de Bruijn come quello della Figura 8.

2.4 Stato dell'Arte: Software VPN

Per cercare di capire quali funzionalità sarebbe stato giusto introdurre in n2n per il raggiungimento dell'obiettivo prefissato, sono stati analizzati software e tecnologie esistenti che fossero in grado di offrire, salvo alcune differenze, lo stesso servizio di n2n, ossia una rete virtuale all'interno della quale gli utenti potessero operare in assoluta sicurezza. In seguito, sono stati analizzati brevemente tre software VPN: Tailscale, ZeroTier e FreePN.

2.4.1 Tailscale

Tailscale [27] è un software in grado di creare una rete sicura tra server, computer, istanze cloud e altri apparati di rete, anche nel caso in cui essi fossero separati da sottoreti o firewalls. Usando Tailscale, si può mettere in comunicazione diretta i dispositivi di una rete, minimizzando la latenza, riducendo la mobilità e utilizzando comunicazioni interamente cifrate (end-to-end).
Questo è un software sviluppato dall'omonima azienda ed è disponibile per Linux, iOS, Windows, Raspberry PI e Android. Si basa principalmente sul software open source WireGuard, il quale crea una serie di tunnel cifrati estremamente leggeri tra un computer, una Virtual Machine (VM), un container e un qualsiasi altro nodo della rete. Una volta avviata una sessione WireGuard, si potrà andare a creare una rete mesh, ossia una rete in cui ogni device è connessa con tutte le altre. Per far questo, servirà avere le chiavi di cifratura di ciascun dispositivo, quindi viene utilizzato un "coordination server", essenzialmente un contenitore condiviso per le chiavi pubbliche degli utenti. Un nodo caricherà quindi la propria chiave pubblica su questo server, dal quale si potrà poi scaricare una lista di chiavi di altri nodi. Tailscale utilizza l'autenticazione a due fattori (2FA) [35] e definisce controlli di accesso basati sui ruoli per limitare l'accesso a server sensibili o autorizzare gli utenti a vedere solo ciò di cui hanno bisogno.

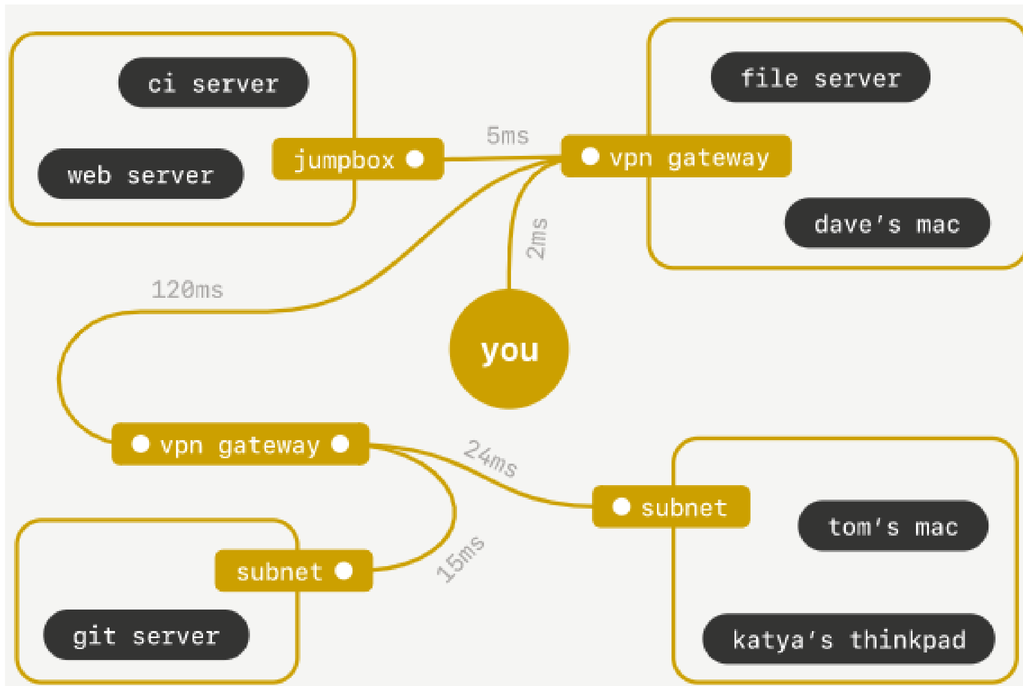

Figura 10: Rete senza l'utilizzo di Tailscale.

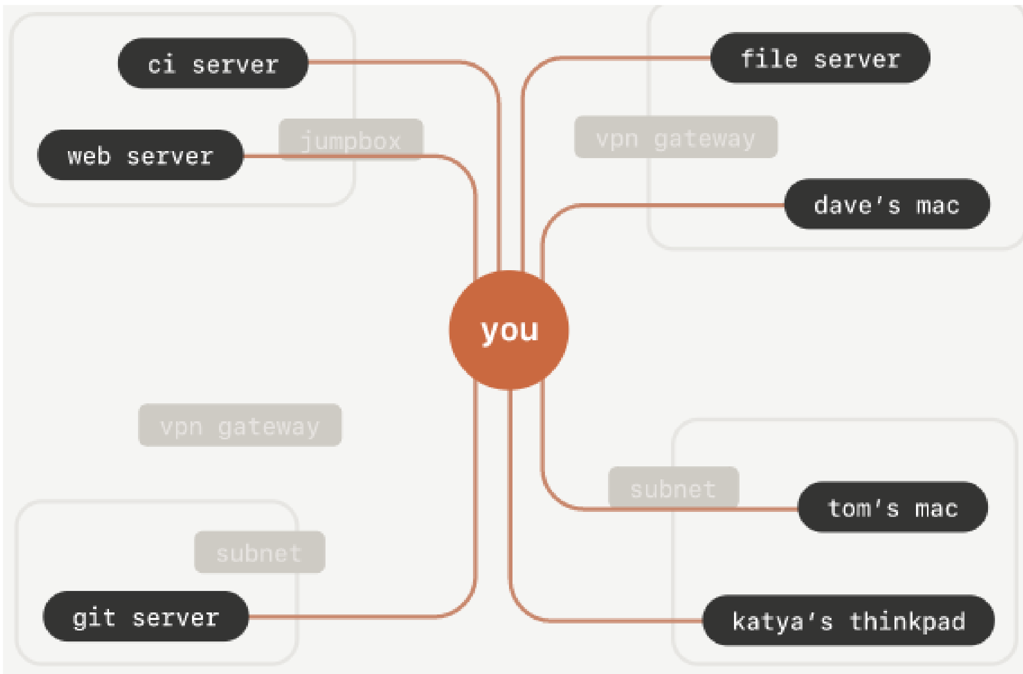

Figura 11: Rete con l'utilizzo di Tailscale.

2.4.2 ZeroTier

ZeroTier [38] è un software sviluppato dall'omonima azienda ed è disponibile per Windows, MacOS, Linux, iOS, Android e FreeBSD. Esso è in grado di connettere i membri di una rete da qualsiasi parte del mondo su qualsiasi dispositivo. ZeroTier crea reti sicure tra dispositivi fissi, cloud, desktop e mobili, e la sua tecnologia di virtualizzazione della rete offre la potenza del networking definito dal software aziendale a ogni dispositivo, servizio e applicazione, sia che si trovi su un dispositivo fisico o nel cloud. Questo software è adatto per le grandi aziende con complesse esigenze di rete, ma allo stesso tempo è abbastanza semplice per essere utilizzato da decine di migliaia di utenti per svolgere azioni elementari, come giocare online o connettersi al proprio PC di casa dal proprio dispositivo mobile.

ZeroTier combina le caratteristiche di VPN (Virtual Private Network) e SD-WAN (Software-defined in Wide Area Network), semplificando la gestione della rete, consente flessibilità emulando il layer 2 del modello ISO/OSI, ha una configurazione rapida e fornisce sicurezza nelle comunicazioni tramite crittografia end-to-end a 256 bit.
Il protocollo del software presenta due livelli: VL1 e VL2. VL1 è il livello di trasporto peer-to-peer sottostante, il "cavo virtuale", mentre VL2 è un livello Ethernet emulato che fornisce ai sistemi operativi e alle app un mezzo di comunicazione familiare. Inoltre, ZeroTier è progettato per essere veloce e a configurazione zero, infatti, un utente potrà avviare un nuovo nodo senza dover scrivere file di configurazione o fornire gli indirizzi IP di altri nodi.
Per raggiungere questo obiettivo VL1 è organizzato come DNS. Alla base della rete c'è una raccolta di root server sempre presenti il cui ruolo è simile a quello dei root nameserver DNS. I root eseguono lo stesso software degli endpoint e hanno come compito la definizione del "mondo". Essa si presenta in due forme: un unico "pianeta", i cui server root sono gestiti da ZeroTier, e una o più "lune". Una "luna" è solo un modo conveniente per aggiungere server root definiti dall'utente, per ridurre la dipendenza dall'infrastruttura di ZeroTier o per individuare i server root più vicini.

La configurazione della connessione peer-to-peer tra due nodi A e B funziona nel modo seguente:

1. A vuole inviare un pacchetto a B, ma poiché non ha un percorso diretto lo invia a monte a R (una radice).
2. Se R ha un collegamento diretto a B, inoltra il pacchetto. Altrimenti invia il pacchetto a monte fino a raggiungere le radici di livello superiore. Esse conoscono tutti i nodi, quindi alla fine il pacchetto raggiungerà B.
3. R invia anche un messaggio chiamato rendezvous ad A contenente suggerimenti su come potrebbe raggiungere B. Nel frattempo la root che inoltra il pacchetto a B invia anch'essa un messaggio rendezvous informando B su come potrebbe raggiungere A.
4. A e B ricevono i loro messaggi e tentano di inviare messaggi di prova l'un l'altro. Se ciò funziona, viene stabilito un collegamento diretto.

2.4.3 FreePN

FreePN [12] è un software open-source sviluppato per Ubuntu e Gentoo e presto disponibile anche su altri sistemi operativi e loro derivati, come Windows, MacOS, Android, Debian e Arch Linux. Questo software fornisce una rete completamente gratuita e dedicata alla protezione della privacy online degli utenti. L'ingresso in rete è facile poiché basta scaricare e installare l'app. Non è richiesto un account, una registrazione, o una configurazione: nel momento in cui l'utente avvia l'app, la protezione è attiva.

FreePN non presenta nessun limite di larghezza di banda: a differenza di altre VPN fornite da altri provider, esso non bloccherà mai la connessione. Per quanto riguarda la sicurezza, questo software utilizza la crittografia AES-256 su tutto il traffico di rete, quindi fintanto che un utente è connesso avrà la sicurezza che le proprie informazioni siano protette. Un'altra importante caratteristica di FreePN è la sua garanzia di essere totalmente anonimo. Costruita sui principi P2P di Tor [37], FreePN è l'unica VPN completamente anonima, non registra mai l'IP di un utente né tiene traccia della sua attività.

Il demone di rete FreePN (fpnd) è un'implementazione P2P di una rete privata virtuale distribuita (dVPN) che crea un insieme anonimo di peer. I peer vengono collegati in modo casuale all'avvio e ricollegati a nuovi peer in base alle necessità. FreePN non è una soluzione VPN completa e non richiede l'installazione di chiavi crittografiche o certificati. Il traffico sui collegamenti di FreePN è sempre crittografato, tuttavia, poiché ogni collegamento di rete è indipendente, il traffico deve essere decifrato quando esce da ciascun peer. Quando viene eseguito in modalità "peer", si presume che ogni nodo sia un host non attendibile; quando si esegue in modalità "ad hoc", si può invece assumere che i nodi siano host attendibili.
Alcune limitazioni sono date dal fatto che viene instradato solo il traffico www (http e https) e dns (opzionale), il routing del traffico supporta solo IPv4 e la privacy del DNS dipende interamente dalla configurazione dell'utente.

Capitolo 3

Architettura del Sistema

In questo capitolo è fornita una descrizione dell'architettura del software n2n, introducendo le entità che generalmente sono coinvolte in una tipica sessione e la modalità con cui n2n rende sicure le comunicazioni tra i nodi della rete. Le nuove funzionalità introdotte e le scelte implementative fatte saranno analizzate con maggior dettaglio e opportunamente motivate nel Capitolo 4.

n2n è una rete privata virtuale (VPN) peer-to-peer di livello 2 che consente agli utenti di sfruttare le funzionalità tipiche delle applicazioni P2P a livello di rete anziché a livello di applicazione [7, 19]. Infatti, la tecnologia P2P è utilizzata prevalentemente a quest'ultimo livello e le sue proprietà benefiche sono quindi limitate alla risoluzione di particolari problemi a livello applicativo. Sulla base di queste considerazioni, è stato deciso di sviluppare n2n sui principi del P2P per interconnettere risorse che altrimenti non sarebbero raggiungibili a causa della configurazione della rete. Questo significa che, tramite n2n, gli utenti possono ottenere visibilità IP nativa: ad esempio, due PC appartenenti alla stessa rete n2n possono eseguire il ping a vicenda ed essere raggiungibili con lo stesso indirizzo IP di rete, indipendentemente dalla rete a cui appartengono attualmente. In altre parole, n2n sposta il P2P dall'applicazione al livello di rete. Da questo punto di vista quindi n2n rappresenta un importante passo avanti rispetto alle tradizionali reti P2P perché è trasparente e utilizzabile da tutte le applicazioni senza che esse sappiano della sua presenza.

Le principali caratteristiche di n2n sono le seguenti :

- n2n è una rete privata crittografata di livello 2 basata su un protocollo P2P.
- La crittografia viene eseguita sui nodi edge utilizzando protocolli aperti con chiavi di crittografia definite direttamente dall'utente.
- Ogni utente n2n può appartenere contemporaneamente a più reti, note come comunità.
- I pacchetti n2n vengono cifrati/decifrati solo dai nodi edge, mentre i supernodi effettuano il loro inoltro basandosi su un'intestazione in chiaro, senza ispezionare il contenuto del pacchetto.
- Capacità di attraversare NAT (Network Address Translation) e firewall dall'esterno verso l'interno, in modo che i nodi n2n siano raggiungibili anche se in esecuzione su una reti LAN private.

Una comunità in n2n rappresenta un insieme di host partecipanti che scelgono di essere parte di essa mantenendo le registrazioni con uno o più supernodi. Si noti anche il fatto che la comunità ed il suo funzionamento sono concettualmente simili al tag VLAN ID nello standard 802.1Q. Gli usi di tali comunità sono molteplici e una modellazione siffatta descrive le comunicazioni interne nella maggior parte delle piccole imprese, gruppi di utenti, affiliazioni, ecc. n2n fornisce quindi un'estensione LAN a una comunità mobile: non importa verso quale direzione si sta spostando un host di una comunità o quale mezzo di trasporto utilizza, rimarrà sempre un membro della rete n2n.

Oltre alle proprietà elencate finora, n2n presenta ulteriori differenze da altri approcci [17]:

- A differenza della maggior parte delle reti P2P (come ad esempio Chord [26]) che sono soggette al problema di individuare nodi in un numero limitato di salti, in n2n questo non è uno svantaggio in quanto i nodi della rete sono raggiungibili direttamente o con un salto quando si passa attraverso la comunità n2n.

- L'appartenenza al nodo n2n è statica. I nodi di solito si registrano all'interno di una comunità e vi appartengono fintanto che il nodo è operativo.
- L'obiettivo di n2n non è il file sharing, ma piuttosto consentire agli utenti di comunicare in modo sicuro e localizzarsi mediante indirizzi che siano indipendenti dalla loro posizione fisica.

Inoltre, i vantaggi principali di n2n rispetto alle tecnologie simili analizzate nel Capitolo 2 sono:

- Possibilità di creare una rete privata senza un punto di controllo centrale.
- Scambio diretto di pacchetti, che aumenta l'efficienza della rete e riduce la latenza.
- Codebase estremamente piccola senza dipendenze da software proprietari, quindi può essere inserito in dispositivi piccoli o altre applicazioni.
- Passo in avanti rispetto alle reti P2P a livello applicativo, perché è trasparente e utilizzabile da tutte le applicazioni.

3.1 Supernode

Un supernodo viene utilizzato dai nodi edge all'avvio. Questa entità è fondamentalmente un registro di directory e un router di pacchetti per quei nodi edge che non possono parlare direttamente. I supernodi introducono ulteriori nodi edge e consentono di raggiungerli anche quando essi si trovano dietro NAT simmetrici. Con questa tecnica tutte le richieste dalla stessa sorgente [IP interno:porta] a una destinazione [IP specifico:porta] sono mappate su un [IP esterno:porta]. Se lo stesso host interno invia una richiesta con lo stesso indirizzo di origine e porta a una destinazione diversa, viene utilizzata una mappatura esterna diversa.

Un supernodo in n2n si basa principalmente su due file: src/sn.c e src/sn_utils.c.
In src/sn.c sono definite le funzioni per effettuare il parsing dei parametri inseriti da linea di comando oppure per analizzare il file di configurazione fornito. All'interno di src/sn.c si trova anche il metodo main, il quale dopo aver inizializzato il supernodo, provvede ad analizzare i parametri forniti da linea di comando o il file di configurazione. Dopodiché vengono collegati due socket alla porta locale UDP e alla porta UDP di management. Infine, il supernodo viene avviato chiamando il ciclo di esecuzione presente in src/sn_utils.c.

Nel file src/sn_utils.c sono contenute tutte le funzioni di utility specifiche per il supernodo, come ad esempio la funzione per inizializzare o terminare un supernodo, la funzione per aggiornare la lista degli edge registrati presso di esso, la funzione per rimuovere le comunità inattive, ecc.

Ci sono poi due metodi fondamentali per realizzare il funzionamento del supernodo. Un primo metodo importante è process_udp, il quale prende in esame un datagramma UDP ricevuto ed in base al tipo di messaggio, definito in include/n2n_typedefs.h, decide l'azione da compiere.

L'altro metodo è run_sn_loop, appunto il ciclo di esecuzione del supernodo stesso. Questo ad ogni iterazione azzera l'insieme di file descriptor che conterrà il file descriptor del socket UDP ed il file descriptor del socket collegato alla porta di management. Questi due identificatori verranno poi settati e si attenderà un certo tempo per controllare se uno di essi è pronto per un'operazione di scrittura o lettura tramite la funzione select. Essa restituirà il numero di file descriptor pronti per una tra le operazioni elencate sopra, dopodiché si controllerà quale tra essi è effettivamente pronto tramite una macro opportuna.

Una volta trovato il file descriptor pronto e una volta controllato di aver ricevuto effettivamente un datagramma UDP, viene chiamata la funzione process_udp per gestire il messaggio. Prima dell'inizio dell'iterazione successiva si valuta se ci sono supernodi e/o comunità inattive e si effettua un ordinamento delle comunità basandosi sul numero di pacchetti cifrati ricevuti.

3.2 Edge

Poiché n2n è una VPN di livello 2, i nodi edge sono identificati in modo univoco da un indirizzo MAC a 48 bit e da un indirizzo e un nome di comunità di 128 bit. I nodi edge vengono eseguiti su hosts situati in LAN pubbliche o private. All'avvio, un edge avrà un elenco di supernodi a cui si registrerà. I supernodi memorizzeranno le informazioni dei nodi edge e tali informazioni dovranno poi essere periodicamente aggiornate. I nodi edge si registrano al primo supernodo disponibile, dopodiché la registrazione con altri supernodi avviene se il primo non risponde.

I nodi edge parlano tramite interfacce TAP [16] virtuali, ognuna delle quali rappresenta un nodo edge n2n. Ogni PC può avere più interfacce TAP, una per rete n2n, in modo che esso possa appartenere a più comunità in contemporanea. TAP è un driver che consente la creazione di periferiche di rete virtuali, quindi la ricezione e l'invio di pacchetti viene eseguito da programmi software. In particolare, TAP è in grado di simulare un dispositivo Ethernet e quindi utilizza i frame di questo tipo.

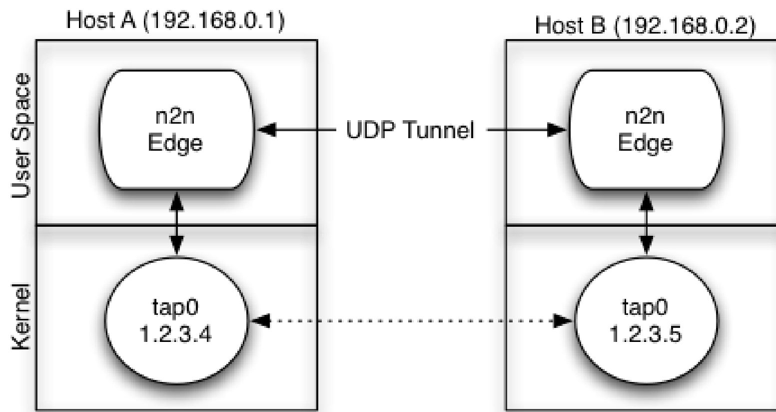

Figura 12: Comunicazione tra due edge in n2n.

Il kernel vede il dispositivo TAP come il percorso attraverso il quale inviare frame Ethernet. I pacchetti UDP che arrivano sul lato IP vengono decifrati ed i frame vengono iniettati nel kernel come se fossero arrivati da un adattatore Ethernet. L'uso di questi dispositivi consente di ridurre quindi il design dell'architettura a concetti familiari.

Quando un nodo edge viene avviato, si registra con il primo supernodo configurato, il quale risponde con un pacchetto di avvenuta registrazione (ack). Se quest'ultimo messaggio non viene ricevuto, il nodo edge passa ad un altro supernodo e proverà nuovamente a registrarsi. Ogni supernodo mantiene un elenco di percorsi a ciascun nodo edge, il quale è codificato dalla coppia [community, indirizzo MAC]. Dato che i nodi edge ricevono anche pacchetti remoti, essi costruiscono a loro volta un elenco di coppie [indirizzo MAC, socket UDP] per i nodi di una comunità, in modo tale da inviare richieste direttamente ad essi.
I nodi edge di n2n possono avere sia indirizzi IP dinamici (ad esempio tramite DHCP) che indirizzi IP statici. Inoltre, la risoluzione dei nomi dei nodi viene effettuata dal servizio DNS (Domain Name System).

La registrazione tra peer fornisce un meccanismo per consentire ai nodi edge di formare connessioni dirette rimuovendo il supernodo dal percorso. Se, invece, uno dei due nodi è dietro NAT simmetrico, l'atto di inviare una richiesta di registrazione apre un percorso di ritorno attraverso il firewall. Se entrambi i peer sono dietro NAT simmetrico, la connettività diretta non è possibile. Come accade con ARP (Address Resolution Protocol) [22], i nodi registrati vengono rimossi nel momento in cui viene rilevato un livello di inattività prolungato per un certo periodo di tempo. Spesso, la registrazione potrebbe non riuscire a causa della presenza di un firewall, in questo caso si potrà utilizzare il routing asimmetrico, ad esempio da edgeB a edgeA via supernodo, ma con una comunicazione diretta nel senso opposto da edgeA ad edgeB (Figura 13).

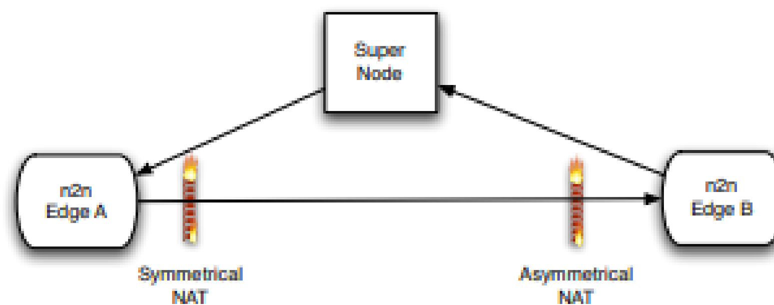

Figura 13: Comunicazione tra due edge in n2n attraverso NAT.

3.3 Crittografia in n2n

Ogni comunità n2n ha una chiave condivisa che viene utilizzata per cifrare/decifrare il payload dei pacchetti. Se un supernodo è compromesso, il traffico in entrata verrà scartato poiché i supernodi non conoscono mai le chiavi di cifratura utilizzate da una comunità. Quando la crittografia del payload è abilitata, il supernodo non sarà più in grado di decifrare il traffico scambiato tra due nodi edge. Riguardo la crittografia del payload di un pacchetto, in n2n sono disponibili quattro versioni diverse di cifrari, i quali possono essere abilitati utilizzando un'opzione apposita dalla riga di comando.

Cipher	Mode	Block Size	Key Size	IV length
Twofish	CTS	128 bits	256 bit	128 bit
AES	CTS	128 bits	128, 192, 256 bit	128 bit
ChaCha20	CTR	Stream	256 bit	128 bit
SPECK	CTR	Stream	256 bit	128 bit

Figura 14: Tabella dei cifrari supportati in n2n.

Twofish [24] antepone un valore casuale a 128 bit al testo normale. Poiché CTS (Ciphertext Stealing) utilizza la modalità CBC (Cipher Block Chaining) sottostante, questo meccanismo ha sostanzialmente lo stesso effetto di un IV (Initialization Vector) più breve. Twofish non richiede riempimento in quanto utilizza uno schema CBC/CTS che può inviare testi cifrati di lunghezza normale.

AES [2] è il cifrario di default utilizzato in n2n e, come Twofish, antepone un valore casuale al testo in chiaro. AES usa uno schema CBC/CTS che può inviare testi cifrati della lunghezza di testo normale. Tuttavia è più lento dei cifrari a flusso poiché la modalità CBC non può competere con essi. Le varianti di AES vengono attivate dalla lunghezza della chiave fornita: con 22 caratteri o meno si usa AES-128, tra 23 e 32 caratteri portano ad AES-192 e 33 o più caratteri attivano AES-256.

ChaCha20 [3] è stato il primo cifrario a flusso supportato da n2n e, oltre all'implementazione C di base, offre una versione SSE (Server Side Encryption). L'intero IV casuale a 128 bit viene trasmesso in chiaro. ChaCha20 di solito è più veloce di AES-CTS.

SPECK [1] è raccomandato dalla NSA (National Security Agency) per uso ufficiale nel caso in cui l'implementazione di AES non sia fattibile a causa di vincoli di sistema (prestazioni, dimensioni, ...). Il cifrario a blocchi viene utilizzato in modalità CTR (Counter) rendendolo un cifrario a flusso. L'intero IV casuale a 128 bit viene trasmesso in chiaro.

Una prova delle prestazioni dei cifrari precedentemente elencati si può ottenere tramite tools/n2n-benchmark. L'intestazione di un pacchetto consiste in una sezione COMMON seguita da una sezione specifica in base al tipo di pacchetto (ad esempio REGISTER, REGISTER_ACK, PACKET, REGISTER_SUPER, ecc.). La sezione COMMON è costituita dai campi illustrati nella Figura 15.

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
! Version=3     ! TTL           ! Flags                         !
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
! Community                                                     :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
! ... Community ...                                             :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
! ... Community ...                                             :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
! ... Community                                                 !
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figura 15: Sezione COMMON di un pacchetto n2n.

Se l'utente decide di abilitare l'apposito flag (-H) per la crittografia dell'intestazione, tutti i campi di un pacchetto vengono crittografati utilizzando il cifrario SPECK in modalità CTR. Si noti che il payload del pacchetto verrà gestito separatamente e non sarà influenzato, poiché la sua cifratura viene eseguita abilitando uno dei cifrari precedentemente elencati. Gli header dei pacchetti devono essere decodificabili dal supernodo, per questo motivo il nome della comunità opera da chiave, perché è già noto al supernodo ed in questo modo non viene introdotta un'ulteriore chiave. Il nome della comunità è composto da un massimo di 16 caratteri, quindi la dimensione della chiave di 128 bit è una scelta ragionevole. Lo schema applicato cerca di mantenere la compatibilità con il formato del pacchetto corrente e funziona come segue:

- La prima riga di 4 byte (versione, TTL, flag) va alla quinta riga.
- Per consentire ad un supernodo di identificare un'intestazione decifrata correttamente in seguito, viene inserito un magic number nella quarta riga. Si usa la stringa ”n2n” e si aggiunge la lunghezza dell'intestazione per poter fermare la decifrazione dell'intestazione subito prima dell'inizio di un eventuale payload.
- Il resto del campo della comunità viene convertito in un IV per la crittografia dell'intestazione.
- Dato che è utilizzato un cifrario a flusso, IV dovrebbe essere un nonce, cioè un valore casuale o pseudocasuale da utilizzare un'unica volta. Per motivi di cifratura e decifrazione dell'header, 32 bit contenenti ”n2n!” vengono concatenati all'IV a 96 bit rendendolo un IV a 128 bit completo.
- La cifratura termina alla fine dell'intestazione. Non comprende il payload che ha il proprio schema di crittografia.

```
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
! IV ...                                                        :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
! ... IV ...                                                    :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
! ... IV                                                        :
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
! 24-bit Magic Number, "n2n" = 0x6E326E         ! Header Length !
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
! Version=3     ! TTL           ! Flags                         !
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figura 16: Header cifrato di un pacchetto n2n.

L'intero pacchetto, incluso il payload, viene sottoposto a checksum utilizzando una funzione di Pearson hashing [36]. Il checksum è una sequenza di bit che, associata al pacchetto trasmesso, aiuta a verificarne l'integrità. Esso viene eseguito sia dai nodi edge che dal supernodo. Anche un timestamp a 52 bit che mostra una precisione al microsecondo è codificato nel IV a 96 bit. A causa del timestamp codificato, l'IV sarà molto probabilmente unico, vicino a un vero nonce [30]. La verifica del timestamp avviene in due fasi:

- Il timestamp (remoto) viene confrontato con l'orologio locale del nodo e non potrà differire di più/meno 16 secondi. Questo limite può essere regolato modificando la definizione TIME_STAMP_FRAME. Quindi, i nodi edge e il supernodo devono mantenere un tempo attuale.
- I timestamp (remoti) validi vengono memorizzati in last_valid_timestamp su ciascun nodo (supernodo e nodi edge). Quindi, il timestamp di un nuovo pacchetto in arrivo può essere confrontato con l'ultimo valido e dovrà risultare uguale o superiore.

I pacchetti di livello 2 vengono anche compressi utilizzando l'algoritmo di Lempel-Ziv-Oberhumer (LZO) [21] il quale è veloce ed efficiente. L'intestazione del pacchetto n2n non è compressa per facilitare l'operazione di inoltro ai supernodi.

Capitolo 4

Implementazione del Software

Come già spiegato nel Capitolo 3, le entità di n2n sono i supernodi e i nodi edge. Una semplice sessione che comprende un supernodo e due nodi edge potrà essere avviata eseguendo i comandi in Figura 17 su host diversi.

On host1 run:

```
$ sudo edge -c mynetwork -k mysecretpass -a 192.168.100.1 -f -l supernode.ntop.org:7777
```

On host2 run:

```
$ sudo edge -c mynetwork -k mysecretpass -a 192.168.100.2 -f -l supernode.ntop.org:7777
```

Figura 17: Comandi da eseguire per avviare una semplice sessione n2n con due nodi edge (host1 e host2) ed un supernodo (supernode.ntop.org).

Entrambi i nodi edge appartengono alla stessa comunità e hanno la stessa chiave di cifratura, in questo modo saranno in grado di comunicare direttamente oppure tramite l'ausilio del supernodo. Il problema del paradigma di comunicazione attuale però, come già spiegato nel Capitolo 1, è dato dal fatto che esso è semi-centralizzato: anche in uno scenario semplice come questo, se il supernodo presso il quale i due nodi edge si sono registrati smettesse di funzionare, essi potrebbero non essere più in grado di comunicare e verrebbero perse le loro configurazioni di rete.

Supponendo di avere uno scenario molto più grande, con tanti dispositivi connessi, tra cui anche entità critiche o con comportamenti dinamici, è facile comprendere che i problemi dovuti ad un paradigma di comunicazione come quello visto finora non potrebbero far altro che aumentare. Questo è il motivo principale per cui si è voluto ridefinire il paradigma di comunicazione di n2n, passando da uno semi-centralizzato ad uno interamente distribuito.

L'approccio seguito è stato quindi quello di studiare inizialmente il codice di n2n, disponibile su GitHub [20]. Dopodiché, dato che n2n è un software open-source, c'è stato un confronto con gli altri sviluppatori a proposito delle funzionalità da introdurre per passare ad un paradigma di comunicazione distribuito. Per ciascuna funzionalità introdotta e per ogni modifica apportata è stata aperta un'apposita issue su GitHub, ossia una sorta di ticket che consente di definire in modo chiaro un bug, un problema o nuova feature, permette di circostanziare la discussione tra i vari sviluppatori e aiuta a tenere traccia dei cambiamenti apportati.

Nel seguito di questo capitolo saranno illustrate quindi le nuove funzionalità introdotte, facendo distinzione tra supernodi e nodi edge. Per ciascuna modifica fatta sarà fornito in dettaglio il suo funzionamento, mostrando anche porzioni di codice, e una o più motivazioni che hanno portato a svilupparla.

4.1 Software: Supernode

Il problema principale di un supernodo è dovuto al fatto che esso non comunica con gli altri supernodi, quindi non conoscerà la situazione circostante, ma saprà solamente chi sono i nodi edge registrati presso di lui. L'idea di base a seguito della quale sono state sviluppate le modifiche da introdurre per un supernodo è data dal fatto che questa entità dovrebbe essere in grado di comunicare con altri supernodi della rete al fine di scambiarsi informazioni. Così facendo un supernodo potrà avere abbastanza informazioni sugli altri, in modo tale da poterle inviare, periodicamente o su richiesta, ai nodi edge, i quali poi gestiranno queste informazioni secondo una loro politica opportuna.

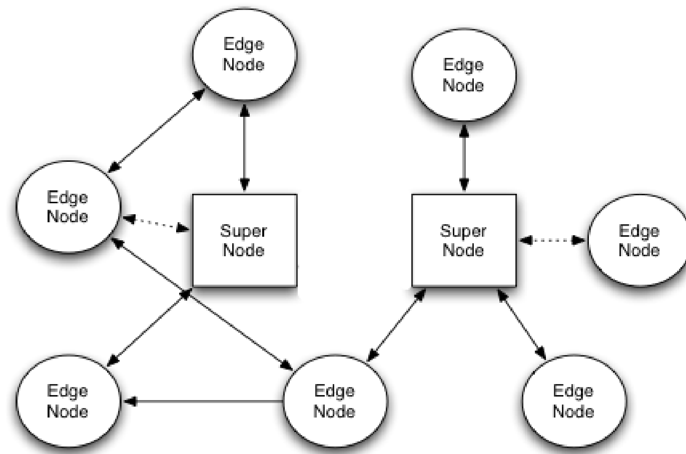

Figura 18: Architettura di n2n.

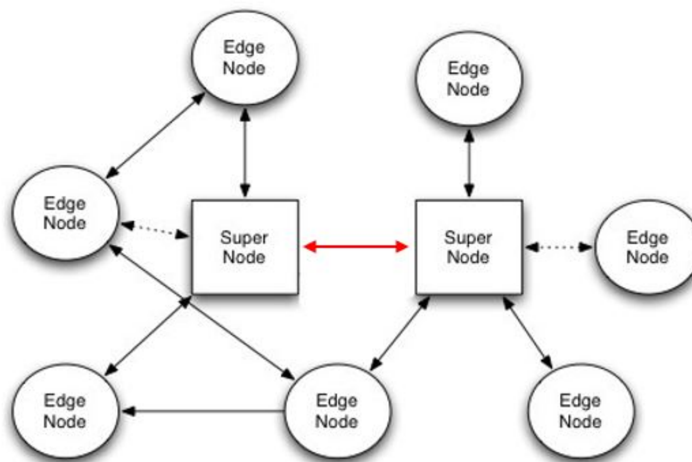

Figura 19: Architettura di n2n con la funzionalità di comunicazione tra supernodi.

Come si vede dalla Figura 18 e dalla Figura 19, la differenza tra le due architetture di n2n a livello visivo è minima, ossia una linea rossa che rappresenta la comunicazione tra i due supernodi. Andando però ad analizzare i cambiamenti introdotti dall'aggiunta di questa funzionalità nel comportamento di un supernodo, si capisce che i possibili malfunzionamenti di questa entità potranno essere gestiti e attenuati.

Sfruttando il meccanismo di registrazione già esistente per i nodi edge e rivedendo la struttura dei messaggi scambiati durante questo processo, un supernodo si potrà quindi registrare presso un altro inviando un messaggio di REGISTER_SUPER. Nel momento in cui la registrazione è andata a buon fine, l'altro supernodo invia un messaggio di risposta REGISTER_SUPER_ACK il cui payload conterrà informazioni (indirizzo MAC e socket) sugli eventuali altri supernodi appartenenti alla rete. Ampliando quindi lo scenario, in una rete più popolata, se un supernodo si può registrare presso un altro, potrà venire a conoscenza degli altri supernodi della rete e comunicare questa informazione agli edge. Se queste informazioni venissero mantenute aggiornate, è chiaro che si avrebbe una funzionalità in grado di fornire uno stato della rete continuamente attendibile e, se ad esempio si verificasse un guasto su un supernodo, i nodi edge registrati presso di lui potrebbero valutare di registrarsi ad un altro supernodo nella rete in base ad un'opportuna strategia di selezione tra i supernodi disponibili che essi conoscono.
Nel seguito sono illustrate quindi nel dettaglio le funzionalità principali introdotte per i supernodi:

- Una comunità di supernodi detta "federazione" che consente ai supernodi di registrarsi tra loro e comunicare.
- La gestione della ricezione di un messaggio REGISTER_SUPER_ACK come conseguenza di un messaggio REGISTER_SUPER inviato da un supernodo.
- Il processo che analizza le informazioni ottenute da un supernodo e verifica se esse sono attendibili per mantenere lo stato della rete aggiornato.

4.1.1 Federazione di supernodi

Per consentire la comunicazione tra supernodi è stata creata un'apposita comunità speciale, detta federazione, all'interno della quale i supernodi potranno scambiarsi messaggi e informazioni così come fanno i nodi edge di una comunità regolare.

Nel file include/n2n_typedefs.h, all'interno della struttura di un supernodo, è stato aggiunto un campo *federation* il quale rappresenta appunto la federazione dei supernodi e viene inizializzato all'interno della funzione di inizializzazione di un supernodo, cioè sn_init. Questo nuovo campo è di tipo sn_community, perché vuole rappresentare una comunità speciale, e viene utilizzato anche per avere un riferimento costante alla federazione, senza bisogno di doverla andare a cercare all'interno della lista delle comunità nel caso in cui ce ne fosse bisogno. In più, all'interno della struttura di una comunità, è stato aggiunto un campo *is_federation* che rappresenta un flag con il quale si potrà controllare se una certa comunità è la federazione dei supernodi o una comunità regolare di nodi edge.

```
struct sn_community *federation;
```

Figura 20: Campo aggiunto alla struttura di un supernodo.

```
uint8_t                is_federation;
```

Figura 21: Campo aggiunto alla struttura di una comunità.

Il nome di default della federazione è "*Federation": è stato inserito un carattere speciale davanti al nome poiché esso è uno dei caratteri, presenti nel file community.list, che non sono ammessi in nomi di comunità regolari. Includendo quindi un carattere speciale all'interno del nome della federazione, si riesce ad evitare il caso in cui un nodo edge utilizzi un nome di comunità uguale al nome della federazione. Quest'ultimo potrà anche essere fornito dalla riga di comando utilizzando l'apposita opzione "-F". All'interno del caso specifico in setOption, viene acquisito il nome passato dalla linea di comando, si aggiunge all'inizio il carattere speciale e si aggiorna il campo della federazione.

```
case 'F': { /* federation name */

    snprintf(sss->federation->community,N2N_COMMUNITY_SIZE-1,"*%s",_optarg);
    sss->federation->community[N2N_COMMUNITY_SIZE-1] = '\0';

  break;
}
```

Figura 22: Caso di setOption che gestisce l'argomento dell'opzione -F.

Per realizzare completamente le funzionalità della federazione di supernodi, sono stati aggiunti un nuovo campo alla struttura di un supernodo e una nuova opzione da riga di comando.

Il nuovo campo inserito in n2n_sn_t è *mac_addr* e memorizza l'indirizzo MAC di un supernodo. Questo indirizzo verrà generato casualmente ed assegnato al supernodo all'interno della funzione sn_init. L'indirizzo MAC di un supernodo serve per distinguere i vari supernodi membri della federazione e viene utilizzato come chiave all'interno della hash table. Infatti, per cercare un supernodo all'interno della struttura, si utilizza il suo indirizzo MAC e, se la macro HASH_FIND restituisce null, significa che il supernodo non è presente.

L'opzione aggiunta è ”-l” e permette, come per i nodi edge, di inserire dalla riga di comando uno o più supernodi con le rispettive porte UDP. Questa opzione è stata aggiunta al supernodo in modo tale che, al momento del suo avvio, esso possa fornire l'indirizzo o il nome di uno o più supernodi già noti a cui registrarsi. L'opzione ”-l” viene gestita in un caso specifico in setOption: dopo aver fatto il controllo sul suo argomento, si chiama la funzione *add_sn_to_list_by_mac_or_sock*. Questa funzione è stata scritta appositamente per gestire questo caso e, data la sua utilità, è stato deciso di utilizzarla anche in altre porzioni di codice come funzione di ricerca di un supernodo o di un edge all'interno della lista. Il motivo principale per cui è stata definita questa funzione deriva dal fatto che l'indirizzo MAC di un supernodo passato con l'opzione

”-l” non sarà noto a priori, quindi inizialmente questo campo sarà settato a zero (indirzzo MAC nullo). All’interno della funzione si realizza una ricerca per indirizzo MAC solamente se esso è diverso dall’indirizzo nullo, altrimenti si verifica se quel supernodo è già presente cercandolo tramite il suo socket. Se la ricerca non ha successo, viene creato un nuovo supernodo, il quale viene aggiunto alla federazione. Quelle che vengono definite come liste, ad esempio la lista dei supernodi, sono in realtà tabelle hash poiché le strutture dati sono definite come ”hashable” e viene utilizzata la libreria uthash [14].

```
if(sss->federation != NULL) {

  skip_add = SN_ADD;
  anchor_sn=add_sn_to_list_by_mac_or_sock(&(sss->federation->edges),socket,(n2n_mac_t*)null_mac,&skip_add);

  if(anchor_sn != NULL){
    anchor_sn->ip_addr = calloc(1,N2N_EDGE_SN_HOST_SIZE);
    if(anchor_sn->ip_addr){
      strncpy(anchor_sn->ip_addr,_optarg,N2N_EDGE_SN_HOST_SIZE-1);
      memcpy(&(anchor_sn->sock), socket, sizeof(n2n_sock_t));
      memcpy(&(anchor_sn->mac_addr),null_mac,sizeof(n2n_mac_t));
      anchor_sn->purgeable = SN_UNPURGEABLE;
      anchor_sn->last_valid_time_stamp = initial_time_stamp();

    }
  }
}
```

Figura 23: Caso di setOption in cui si gestisce l’argomento dell’opzione -l.

```
if(memcmp(mac,null_mac,sizeof(n2n_mac_t)) != 0) { /* not zero MAC */
  HASH_FIND_PEER(*sn_list, mac, peer);
}

if(peer == NULL) { /* zero MAC, search by socket */
  HASH_ITER(hh,*sn_list,scan,tmp) {
          if(memcmp(&(scan->sock), sock, sizeof(n2n_sock_t)) == 0) {
            HASH_DEL(*sn_list, scan);
            memcpy(&(scan->mac_addr), mac, sizeof(n2n_mac_t));
            HASH_ADD_PEER(*sn_list, scan);
            peer = scan;
            break;
          }
  }

  if((peer == NULL) && (*skip_add == SN_ADD)) {
          peer = (struct peer_info*)calloc(1,sizeof(struct peer_info));
           if(peer) {
             sn_selection_criterion_default(&(peer->selection_criterion));
             memcpy(&(peer->sock),sock,sizeof(n2n_sock_t));
             memcpy(&(peer->mac_addr),mac, sizeof(n2n_mac_t));
             HASH_ADD_PEER(*sn_list, peer);
             *skip_add = SN_ADD_ADDED;
           }
  }
}
```

Figura 24: Corpo della funzione *add_sn_to_list_by_mac_or_sock*.

4.1.2 REGISTER_SUPER_ACK Payload

Con la creazione di una federazione, i supernodi sono in grado di comunicare tra di loro e di scambiarsi messaggi. In particolare, per consentire ad un supernodo di registrarsi presso un altro, si potrà utilizzare il messaggio esistente REGISTER_SUPER e la risposta REGISTER_SUPER_ACK con le opportune modifiche.

```
typedef struct n2n_REGISTER_SUPER {
  n2n_cookie_t        cookie;
  n2n_mac_t           edgeMac;
  n2n_sock_t          sock;
  n2n_ip_subnet_t     dev_addr;
  n2n_desc_t          dev_desc;
  n2n_auth_t          auth;
} n2n_REGISTER_SUPER_t;
```

```
typedef struct n2n_REGISTER_SUPER_ACK{
  n2n_cookie_t        cookie;
  n2n_mac_t           edgeMac;
  n2n_ip_subnet_t     dev_addr;
  uint16_t            lifetime;
  n2n_sock_t          sock;
  uint8_t             num_sn;


} n2n_REGISTER_SUPER_ACK_t;
```

Figura 25: Formato dei pacchetti REGISTER_SUPER e REGISTER_SUPER_ACK.

Un supernodo riceve quindi un pacchetto REGISTER_SUPER e controlla la sua provenienza: esso non riuscirà a distinguere se il messaggio proviene da un supernodo della federazione e lo interpreterà come se fosse proveniente da una comunità regolare. La differenza che realizza il funzionamento corretto si basa sul flag *from_supernode*, che è settato a 1 quando il messaggio proviene da un supernodo. In questo modo si renderà necessario un confronto tra questo flag e il flag *is_federation* di una comunità: se il messaggio ricevuto proviene da una comunità regolare (entrambi i flag a zero) o da un altro supernodo della federazione (entrambi i flag a uno), la gestione del pacchetto continuerà. In tutti gli altri casi il pacchetto verrà scartato.

Dopo i precedenti controlli, il supernodo che ha ricevuto un pacchetto REGISTER_SUPER costruisce un pacchetto di risposta REGISTER_SUPER_ACK all'interno del quale verranno inserite le informazioni sui supernodi conosciuti della rete. Per costruire il payload (Figura 26) del messaggio di risposta è stato utilizzato un ciclo (Figura 27) che va ad iterare tra i supernodi nella federazione. Nel payload verranno aggiunti i supernodi fintanto che ci sarà spazio, escludendo il supernodo stesso ed i supernodi che risultano inattivi da più di 20 secondi e di cui bisognerà testare lo stato.

```c
/* REGISTER_SUPER_ACK may contain extra payload (their number given by num_sn)
 * of following type describing a(nother) supernode */
typedef struct n2n_REGISTER_SUPER_ACK_payload {
  n2n_sock_t          sock;          /**< socket of supernode */
  n2n_mac_t           mac;           /**< MAC of supernode */
} n2n_REGISTER_SUPER_ACK_payload_t;
```

Figura 26: Struttura del payload di REGISTER_SUPER_ACK.

```c
payload = (n2n_REGISTER_SUPER_ACK_payload_t*)payload_buf;
HASH_ITER(hh, sss->federation->edges, peer, tmp_peer) {
  if(skip){
    skip--;
    continue;
  }
  if(memcmp(&(peer->sock), &(ack.sock), sizeof(n2n_sock_t)) == 0) continue;
  if((now - peer->last_seen) >= (2*LAST_SEEN_SN_ACTIVE)) continue;
  if(((++num)*REG_SUPER_ACK_PAYLOAD_ENTRY_SIZE) > REG_SUPER_ACK_PAYLOAD_SPACE) break;
  memcpy(&(payload->sock), &(peer->sock), sizeof(n2n_sock_t));
  memcpy(&(payload->mac),  &(peer->mac_addr), sizeof(n2n_mac_t));
  // shift to next payload entry
  payload++;
}
```

Figura 27: Ciclo di assemblaggio del payload di REGISTER_SUPER_ACK.

Sono state poi introdotte delle modifiche alla gestione della ricezione di un pacchetto REGISTER_SUPER_ACK da parte di un supernodo. Anche in questo caso si va a controllare se i flag from_supernode e is_federation corrispondono e, in caso affermativo, si procede alla decifrazione del pacchetto e all'analisi dei suoi campi. Se il pacchetto ricevuto proviene da un supernodo, si andrà ad aggiornare il suo campo last_seen in modo da dimostrare che quel supernodo è attivo e sta inviando dei messaggi, dopodiché, con un altro ciclo simile al precedente, si itera sul payload di REGISTER_SUPER_ACK per acquisire le informazioni ricevute e memorizzarle (Figura 28).
In particolare, tramite la funzione *add_sn_to_list_by_mac_or_sock* si cerca, all'interno della federazione, i supernodi di cui sono state ricevute le informazioni, i quali saranno aggiornati o aggiunti se non presenti. Qualora un supernodo non fosse presente, viene aggiunto ed il suo campo last_seen viene inizializzato con un valore apposito poiché non ci sarà la sicurezza che quel supernodo sia attivo, per cui dovrà essere testato.

```
for(i=0; i<ack.num_sn; i++) {
  skip_add = SN_ADD;
  tmp = add_sn_to_list_by_mac_or_sock(&(sss->federation->edges), &(payload->sock), &(payload->mac), &skip_add);

  if(skip_add == SN_ADD_ADDED) {
    tmp->last_seen = now - LAST_SEEN_SN_NEW;
  }

  // shift to next payload entry
  payload++;
}
```

Figura 28: Ciclo di ispezione e memorizzazione del payload di REGISTER_SUPER_ACK.

In Figura 29 è mostrato un diagramma temporale di come si evolve nel tempo lo scambio di messaggi REGISTER_SUPER e REGISTER_SUPER_ACK tra un edge ed un supernodo e tra due supernodi. Come si può vedere, lo schema adottato per la registrazione tra supernodi è identico allo schema già esistente attuato dai nodi edge.

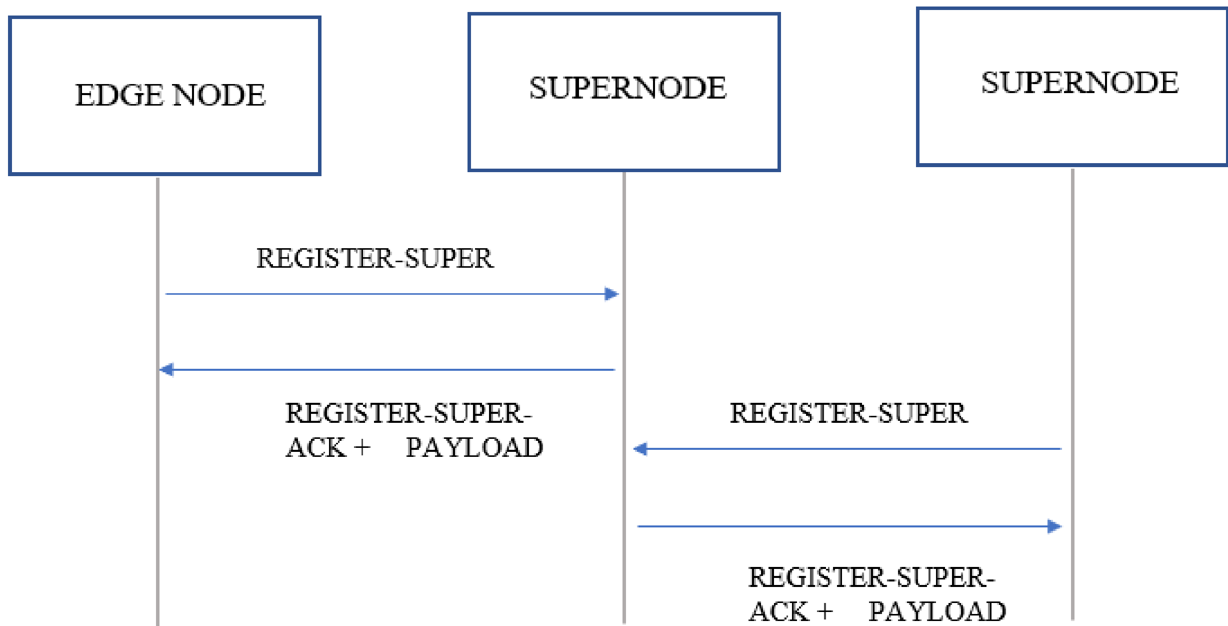

Figura 29: Esempio di un nodo edge e un supernodo che si registrano entrambi presso lo stesso supernodo tramite un messaggio REGISTER_SUPER. Il messaggio di risposta REGISTER_SUPER_ACK conterrà le informazioni sui supernodi conosciuti (indirizzo MAC e socket).

4.1.3 Processo di re-registrazione ed eliminazione

Un problema nel ricevere una lista di supernodi da un pacchetto REGISTER_SUPER_ACK è dato dal fatto che il payload dovrà essere controllato e non potrà essere preso per buono. Infatti, un supernodo presente nel payload del pacchetto potrebbe non essere più attivo, quindi se non fosse controllata la veridicità delle informazioni ricevute ci potrebbe essere il rischio in futuro di trasmettere alla rete informazioni non del tutto corrette.

Per questo è stata definita, in src/sn_utils.c, una funzione re_register_and_purge_supernodes che scorre i supernodi della federazione e controlla il loro stato. Dal controllo potranno scaturire tre stati diversi in cui un supernodo si può trovare:

- *Corrente*: un supernodo corrente è attivo, in questo caso vale che now - last_seen < LAST_SEEN_SN_ACTIVE (20 secondi).
- *Non sicuro*: questo stato viene assegnato a quei supernodi che vengono aggiunti a seguito delle informazioni ricevute dal payload di un pacchetto REGISTER_SUPER_ACK. Non sapendo lo stato esatto del supernodo, esso viene aggiunto e viene inizializzato il suo campo last_seen ad un valore apposito in modo tale che non venga considerato corrente né che venga rimosso. I nodi con questo stato invieranno quindi un messaggio di REGISTER_SUPER, riutilizzando il codice già esistente, per dare prova della loro attività. In questo caso now - last_seen < LAST_SEEN_SN_INACTIVE (90 secondi).
- *Inattivo*: un supernodo inattivo avrà un valore molto alto all'interno del proprio campo last_seen, quindi verrà eliminato dalla funzione purge_expired_registrations, presente nel codice. In questo caso now - last_seen >= LAST_SEEN_SN_INACTIVE (90 secondi).

L'invocazione di questa funzione si trova all'interno del ciclo principale run_sn_loop di un supernodo e avviene ogni 10 secondi, in modo tale che le informazioni scambiate tra i supernodi siano costantemente aggiornate e adatte per poter essere condivise con i nodi edge, che a loro volta le useranno in base a politiche interne specifiche.

4.2 Software: Edge

Le funzionalità introdotte lato edge ed i cambiamenti effettuati sono una diretta conseguenza delle modifiche apportate al supernodo. Infatti, così come avviene per la registrazione tra due supernodi, anche quando un edge si registra ad un supernodo tramite un messaggio REGISTER_SUPER, riceverà poi un messaggio di risposta REGISTER_SUPER_ACK con un payload contenente informazioni sui vari supernodi della rete.

Diventa quindi importante implementare anche per i nodi edge una funzionalità che analizzi il payload ricevuto, in maniera tale da verificare le informazioni al suo interno e poterle utilizzare in modo corretto. Tramite le informazioni ricevute da un messaggio REGISTER_SUPER_ACK infatti, un edge sarà in grado di conoscere i supernodi della rete. Questo consente ad un nodo edge di potersi registrare ad uno dei supernodi conosciuti qualora il supernodo originario smettesse improvvisamente di funzionare.

Poiché un nodo edge manteneva solamente gli indirizzi IP dei supernodi in un array, questo campo è stato rimosso dalla struttura di un edge e sostituito con una lista di supernodi, come ad esempio è stato fatto lato supernodo all'interno della federazione. In questo modo, per ogni supernodo è possibile conoscere più informazioni, principalmente il suo indirizzo MAC e il suo socket. Inoltre, sempre all'interno della struttura di un edge, è stato aggiunto un campo *curr_sn* che rappresenta un puntatore al supernodo con cui l'edge è attualmente registrato.

```
struct peer_info    *supernodes;/**< List of supernodes */
```

Figura 30: Lista dei supernodi mantenuta da un nodo edge.

```
struct peer_info    *curr_sn;/**< Currently active supernode. */
```

Figura 31: Puntatore al supernodo presso il quale il nodo edge è attualmente registrato.

Sono state introdotte modifiche anche al blocco di codice di src/edge_utils.c responsabile della gestione di un pacchetto REGISTER_SUPER_ACK. Per fare ciò è stato adattato il codice usato anche per la gestione del payload di REGISTER_SUPER_ACK in src/sn_utils.c. Il programma itera quindi attraverso il payload e chiama la funzione *add_sn_to_list_by_mac_or_sock* per verificare se un supernodo contenuto nel payload è già presente nella lista o se va aggiunto.

```
for(i=0; i<ra.num_sn; i++){
  skip_add = SN_ADD;
  sn = add_sn_to_list_by_mac_or_sock(&(eee->conf.supernodes), &(payload->sock), &(payload->mac), &skip_add);

  if(skip_add == SN_ADD_ADDED){
    sn->ip_addr = calloc(1,N2N_EDGE_SN_HOST_SIZE);
    if(sn->ip_addr != NULL){
      inet_ntop(payload->sock.family,
                (payload->sock.family == AF_INET)?(void*)&(payload->sock.addr.v4):(void*)&(payload->sock.addr.v6),
                sn->ip_addr, N2N_EDGE_SN_HOST_SIZE-1);
      sprintf (sn->ip_addr, "%s:%u", sn->ip_addr, (uint16_t)(payload->sock.port));
    }
    sn_selection_criterion_default(&(sn->selection_criterion));
    sn->last_seen = now - LAST_SEEN_SN_NEW;
    sn->last_valid_time_stamp = initial_time_stamp();
    traceEvent(TRACE_NORMAL, "Supernode '%s' added to the list of supernodes.", sn->ip_addr);
  }
  // shfiting to the next payload entry
  payload++;
}
```

Figura 32: Ciclo di gestione e memorizzazione del payload di REGISTER_SUPER_ACK da parte di un nodo edge.

Una volta definita la modalità con cui un edge memorizza e riceve informazioni sui supernodi della rete, diventa quindi fondamentale stabilire una strategia di selezione. Se ad esempio il supernodo presso il quale l'edge è registrato cessa improvvisamente di funzionare, l'edge dovrà ispezionare la propria lista di supernodi e registrarsi presso un altro in modo tale da continuare a compiere le proprie operazioni. Nel seguito è illustrato come è stata realizzata questa funzionalità e quali sono i concetti di base utilizzati per sviluppare questa strategia di selezione. Inoltre, sarà illustrata più nel dettaglio la strategia adottata in partenza, basata su RTT tra edge e supernodo, e quella scelta attualmente funzionante, basata invece sul carico di lavoro di un supernodo. Infine, sarà fatto un confronto tra le due strategie per spiegare meglio le loro caratteristiche e i loro criteri di scelta.

4.2.1 Strategia di selezione basata su RTT

Una volta che un nodo edge ha ricevuto le informazioni sui supernodi della rete, esse verranno mantenute aggiornate grazie alla continua comunicazione che si ha tra supernodi e nodi edge e tra i supernodi della federazione. Tramite queste informazioni un nodo edge potrà quindi venire a conoscenza della topologia della rete e avere una maggiore capacità di risposta a situazioni critiche.

Riprendendo gli esempi fatti nei capitoli precedenti, se un supernodo con diversi nodi edge registrati smettesse improvvisamente di funzionare, i nodi edge si troverebbero impossibilitati a compiere alcune, se non tutte, le loro operazioni. Ma conoscendo la topologia della rete e le informazioni sui supernodi che la compongono, essi saranno in grado di potersi registrare a nuovi supernodi e continuare le proprie attività.

La scelta del nuovo supernodo a cui registrarsi diventa quindi il punto cruciale di questa funzionalità poiché potrà dipendere da molti fattori, come ad esempio la topologia della rete, lo stato dei supernodi, la latenza tra le comunicazioni delle entità coinvolte, ecc. La prima scelta fatta è stata quella di sviluppare una strategia di selezione semplice in modo tale da poter testare l'effettivo funzionamento di questo nuovo meccanismo, dopodiché questa funzionalità è stata parzialmente cambiata ed è stata sostituita da una strategia leggermente più complessa basata sul carico di lavoro di un supernodo.
Per rendere il codice flessibile ed in grado di poter supportare altre strategie future, senza richiedere una grossa mole di lavoro, la strategia di selezione è stata implementata in un modulo chiamato src/sn_selection.c, le cui funzioni sono state definite nel file include/sn_selection.h. Si è poi aggiunto un nuovo campo, chiamato *selection_criterion*, alla struttura peer_info per rappresentare il tipo di dato sul quale la strategia scelta si basa. Il suo tipo è definito all'interno della macro SN_SELECTION_CRITERION_DATA_TYPE in modo tale che, se in futuro verrà adottata una strategia diversa, si potrà cambiare modificando semplicemente il valore della macro.

Il motivo per cui è stato deciso di implementare la strategia di selezione in un modulo assestante è dato dal fatto che per adesso la strategia supportata è una ed una soltanto. Per questa ragione, in futuro potrebbe nascere il bisogno di sviluppare una nuova strategia che tenga conto di altri fattori: concentrando le funzionalità della strategia in un modulo dedicato ed apposito, un'eventuale cambio o una rimodellazione della strategia adottata richiederà a quel punto poche modifiche in punti del codice ben precisi.

Per realizzare la strategia è stato utilizzato il messaggio QUERY_PEER. Il motivo per cui è stato sfruttato questo messaggio ha a che fare con l'implementazione della prima strategia, quella più semplice. Essa è basata su RTT (Round Trip Time), poiché si utilizza come criterio di selezione il tempo che intercorre tra l'invio di un messaggio e la ricezione della relativa risposta tra un supernodo ed un nodo edge. Per far questo ho utilizzato quindi un messaggio QUERY_PEER in cui il campo targetMac è settato a zero per distinguerlo da un messaggio QUERY_PEER normale. In questo caso QUERY_PEER non viene usato per ottenere informazioni su un dato supernodo ma solo per calcolare il tempo tra il suo invio e la ricezione della risposta PEER_INFO.

```
typedef struct n2n_QUERY_PEER
{
  n2n_mac_t          srcMac;
  n2n_sock_t         sock;
  n2n_mac_t          targetMac;
} n2n_QUERY_PEER_t;
```

Figura 33: Struttura di un messaggio QUERY_PEER.

Il tempo tra l'invio e la risposta verrà calcolato lato edge e verrà memorizzato all'interno del campo selection_criterion del supernodo che è stato testato. In seguito, all'interno di src/edge_utils.c, è stata definita una funzione *sort_supernodes* (Figura 34) che viene chiamata all'interno del ciclo principale dell'edge ogni 30 secondi. Questa funzione ordina i supernodi della lista in base al valore di selection_criterion, dopodiché setta questo campo con una macro di default e invierà nuovamente un messaggio QUERY_PEER per ricalcolare il tempo di invio e ricezione alla prossima iterazione.

Così facendo, la lista di supernodi sarà sempre aggiornata e ordinata. Inoltre, prima di effettuare l'ordinamento, ad ogni iterazione, viene valutato anche se il supernodo a cui un nodo edge è registrato corrisponde al primo supernodo della lista. Se così non fosse, si provvede ad aggiornare il puntatore curr_sn e viene inviato un messaggio di REGISTER_SUPER al primo supernodo della lista, che risulterà essere il più conveniente secondo la strategia adottata basata su RTT.

```
static int sort_supernodes(n2n_edge_t *eee, time_t now){
  struct peer_info *scan, *tmp;

 if(eee->curr_sn != eee->conf.supernodes){
   eee->curr_sn = eee->conf.supernodes;
   memcpy(&eee->supernode, &(eee->curr_sn->sock), sizeof(n2n_sock_t));
   eee->sup_attempts = N2N_EDGE_SUP_ATTEMPTS;

   traceEvent(TRACE_INFO, "Registering with supernode [%s][number of supernodes %d][attempts left %u]",
              supernode_ip(eee), HASH_COUNT(eee->conf.supernodes), (unsigned int)eee->sup_attempts);

   send_register_super(eee);
   eee->sn_wait = 1;
 }

 if(now - eee->last_sweep > SWEEP_TIME){
    if(eee->sn_wait == 0){
      /* sorting supernodes in ascending order
       * basing on their selection_criterion. */
      sn_selection_sort(&(eee->conf.supernodes));
    }

    HASH_ITER(hh, eee->conf.supernodes, scan, tmp){
      sn_selection_criterion_default(&(scan->selection_criterion));
    }
    sn_selection_criterion_common_data_default(eee);

    send_query_peer(eee, null_mac);
    eee->last_sweep = now;
  }

  return 0; /* OK */
}
```

Figura 34: Funzione sort_supernodes chiamata ripetutamente nel ciclo principale di un nodo edge. La funzione di ordinamento effettiva è sn_selection_sort, implementata nel file src/sn_selection.c.

4.2.2 Strategia di selezione basata su carico

Utilizzare una strategia di selezione così semplice come quella basata su RTT potrebbe portare comunque ad alcuni problemi. In particolare, essa non tiene conto del carico di lavoro di un supernodo. Un potenziale rischio è dato dal fatto che un nodo edge potrebbe essere indirizzato verso un supernodo con un RTT molto basso ma con un carico di lavoro elevato, magari con molti nodi edge già registrati. Per questo è necessario che la strategia di selezione scelta e realizzata in n2n si basi sul carico di lavoro di un supernodo, per consentire una distribuzione il più possibile equa del carico della rete nel tempo. In questo modo, infatti, quando un edge dovrà registrarsi ad un nuovo supernodo, non verrà indirizzato verso il supernodo con RTT minore, ma verso quello con il carico di lavoro più basso tra quelli presenti.

Per realizzare questa strategia non sono stati necessari grandi cambiamenti al codice poiché la strategia basata su RTT era già stata implementata e resa funzionante. Le maggiori modifiche sono state apportate al file src/sn_selection.c, che contiene le funzioni ausiliarie per realizzare la strategia ed eventualmente cambiare i suoi criteri di selezione. Un'altra differenza tra la strategia basata su RTT e quella basata su carico consiste nel fatto che in quest'ultima la risposta PEER_INFO avrà un nuovo campo *data* (Figura 35) contenente un valore rappresentante il carico di lavoro del supernodo che ha ricevuto il messaggio QUERY_PEER.

```
typedef struct n2n_PEER_INFO {
  uint16_t              aflags;
  n2n_mac_t             srcMac;
  n2n_mac_t             mac;
  n2n_sock_t            sock;
  SN_SELECTION_CRITERION_DATA_TYPE data;
} n2n_PEER_INFO_t;
```

Figura 35: Struttura di un messaggio PEER_INFO.

Il calcolo del carico di lavoro di un supernodo viene eseguito da una funzione presente nel file src/sn_selection.c. Questa funzione, come si vede nella Figura 36, itera su tutte le comunità supportate di un supernodo e, per ognuna di esse, calcola il numero di nodi edge tramite la macro HASH_COUNT. I nodi edge delle comunità che hanno abilitata la crittografia degli header dei pacchetti vengono contati due volte poiché esercitano un maggior carico di lavoro presso il supernodo.

```
/* Function that gathers requested data on a supernode. */
SN_SELECTION_CRITERION_DATA_TYPE sn_selection_criterion_gather_data(n2n_sn_t *sss){
  SN_SELECTION_CRITERION_DATA_TYPE data = 0, tmp = 0;
  struct sn_community *comm, *tmp_comm;

  HASH_ITER(hh, sss->communities, comm, tmp_comm){
    tmp = HASH_COUNT(comm->edges) + 1; /* number of nodes in the community + the community itself. */
    if(comm->header_encryption == HEADER_ENCRYPTION_ENABLED){
      tmp *= 2;
    }
    data += tmp;
  }

  return data;
}
```

Figura 36: Funzione di src/sn_selection.c che calcola il carico di lavoro di un supernodo.

La funzione mostrata sopra viene invocata lato supernodo al momento della costruzione del pacchetto PEER_INFO ed il suo risultato viene inserito nel campo *data*. Una volta che il nodo edge riceve questo pacchetto, chiama la funzione *sn_selection_criterion_calculate* (Figura 37) per memorizzare il valore ricevuto nel campo apposito del supernodo che ha inviato il messaggio di risposta.

A causa della strategia di rimozione dei nodi, gli edge potrebbero saltare da un supernodo all'altro poiché il loro carico cambia: è stato quindi introdotto un fattore di mitigazione di questi salti. Questo fenomeno potrebbe portare a dei costi aggiuntivi dovuti al cambio di supernodo, quindi se il supernodo rimane lo stesso non ci sarà motivo di cambiare e verrà mitigato il suo carico di lavoro per far sì che all'iterazione successiva venga nuovamente selezionato e si eviti un salto presso un altro supernodo.

```c
int sn_selection_criterion_calculate(n2n_edge_t *eee, peer_info_t *peer, SN_SELECTION_CRITERION_DATA_TYPE *data){
  SN_SELECTION_CRITERION_DATA_TYPE common_data;
  int sum = 0;

  common_data = sn_selection_criterion_common_read(eee);

  peer->selection_criterion = (SN_SELECTION_CRITERION_DATA_TYPE)(be32toh(*data) + common_data);

  /* Mitigation of the real supernode load in order to see less oscillations.
   * Edges jump from a supernode to another back and forth due to purging.
   * Because this behavior has a cost of switching, the real load is mitigated with a stickyness factor.
   * This factor is dynamically calculated basing on network size and prevent that unnecessary switching */
  if(peer == eee->curr_sn){
    sum = HASH_COUNT(eee->known_peers) + HASH_COUNT(eee->pending_peers);
    peer->selection_criterion = peer->selection_criterion * sum / (sum + 1);
  }

  return 0; /* OK */
}
```

Figura 37: Funzione che ottiene il carico di lavoro di un supernodo e lo memorizza nell'apposito campo della struttura peer_info.

Per quanto riguarda l'ordinamento dei supernodi all'interno della lista, esso avviene sempre tramite la funzione sort_supernodes, che dispone la lista in ordine crescente in base al carico di lavoro di un supernodo. Il primo supernodo della lista dopo l'ordinamento risulterà quindi essere quello con meno lavoro da compiere, e quindi il migliore da selezionare per mantenere una equa distribuzione del carico di lavoro tra i supernodi della rete.

4.2.3 Confronto delle strategie

La funzionalità di strategia di selezione di un supernodo è stata originariamente sviluppata basandosi sulla metrica RTT. Questa metrica indica, nelle telecomunicazioni in generale, il tempo che intercorre tra l'invio di un segnale e la ricezione della conferma di quel segnale. Quindi, in questo caso è stato misurato il tempo intercorso tra l'invio di un messaggio QUERY_PEER e la ricezione della sua risposta PEER_INFO. Esso è stato calcolato sul nodo edge ed il risultato è stato memorizzato all'interno del campo *selection_criterion* del supernodo che ha inviato la riposta PEER_INFO.

Una volta calcolato questo valore per tutti i supernodi della rete, essi verranno disposti all'interno della lista in ordine crescente. Così facendo, il primo supernodo della lista dopo l'ordinamento sarà quello che, se contattato dal nodo edge corrente, risponderà in minor tempo.

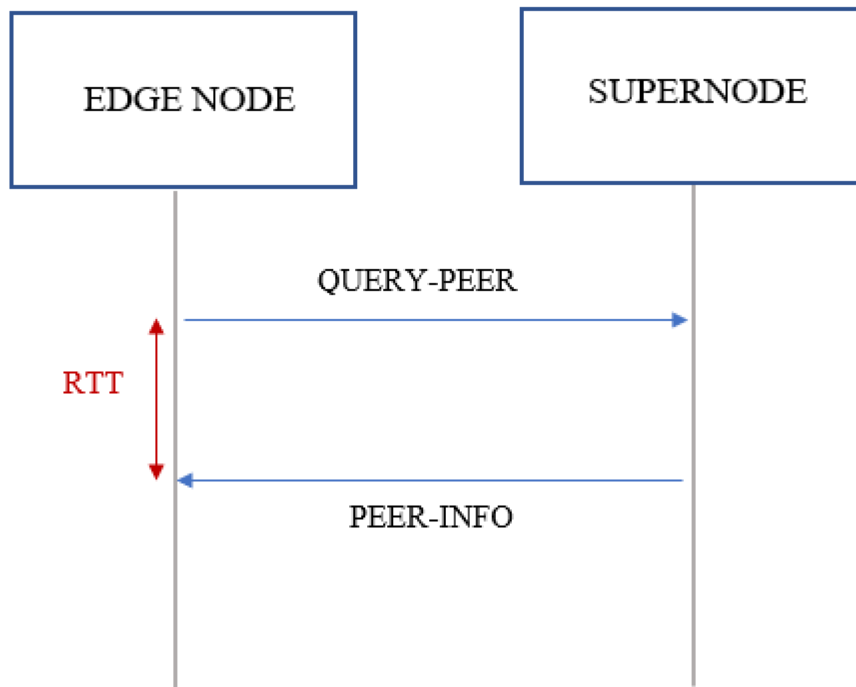

Figura 38: Invio di un messaggio QUERY_PEER e ricezione di un messaggio PEER_INFO nella strategia basata su RTT.

Una strategia di questo tipo è di facile implementazione e non richiede dati aggiuntivi da introdurre nei due messaggi scambiati. Il problema legato all'utilizzo di essa è dato dal fatto che si tiene conto solo della metrica RTT, la quale risulta essere indicativa solo in parte. Infatti, come già detto, potrebbe accadere che utilizzando una strategia di selezione con RTT, un edge venga indirizzato verso un supernodo con un basso valore di RTT ma con un alto carico di lavoro. Anche quest'ultima metrica è molto importante, poiché registrarsi presso un supernodo con un carico di lavoro già elevato potrebbe portare a rallentamenti del servizio o, nei casi peggiori, all'indisponibilità per brevi periodi di tempo. Quindi si è deciso di adottare una strategia diversa, basata sul carico di lavoro di un supernodo.

Per fare ciò non c'è stato bisogno di un grosso cambiamento del codice, poiché si è dovuto solamente modificare le funzioni di src/sn_selection.c. La strategia basata su carico è simile alla precedente, poiché sfrutta i due messaggi QUERY_PEER e PEER_INFO. La differenza principale è data dal fatto che una volta che un supernodo ha calcolato il proprio carico di lavoro, comunica questa informazione al nodo edge inserendola nel campo *data* del messaggio PEER_INFO. Il calcolo del carico di lavoro di un supernodo viene svolto dalla funzione *sn_selection_criterion_gather_data* (Figura 36) la quale calcola, per ogni comunità gestita dal supernodo, il numero di nodi edge presenti. Se la comunità ha abilitata anche la crittografia per le intestazioni dei messaggi, essa e i suoi nodi edge verranno conteggiati due volte, poiché esercitano un carico di lavoro aggiuntivo sul supernodo.

Il risultato del calcolo verrà memorizzato all'interno del campo *data* del messaggio PEER_INFO, il quale viene inviato al nodo edge. Esso, una volta ricevuto il pacchetto, acquisisce il carico di lavoro del supernodo e lo memorizza nella sua struttura, contenuta nella lista. Una volta acquisito il carico di lavoro di tutti i supernodi, la lista verrà disposta in ordine crescente in base al valore del carico di lavoro. Il primo supernodo della lista dopo l'ordinamento sarà quindi il supernodo con meno nodi edge connessi e quindi meno lavoro da compiere.

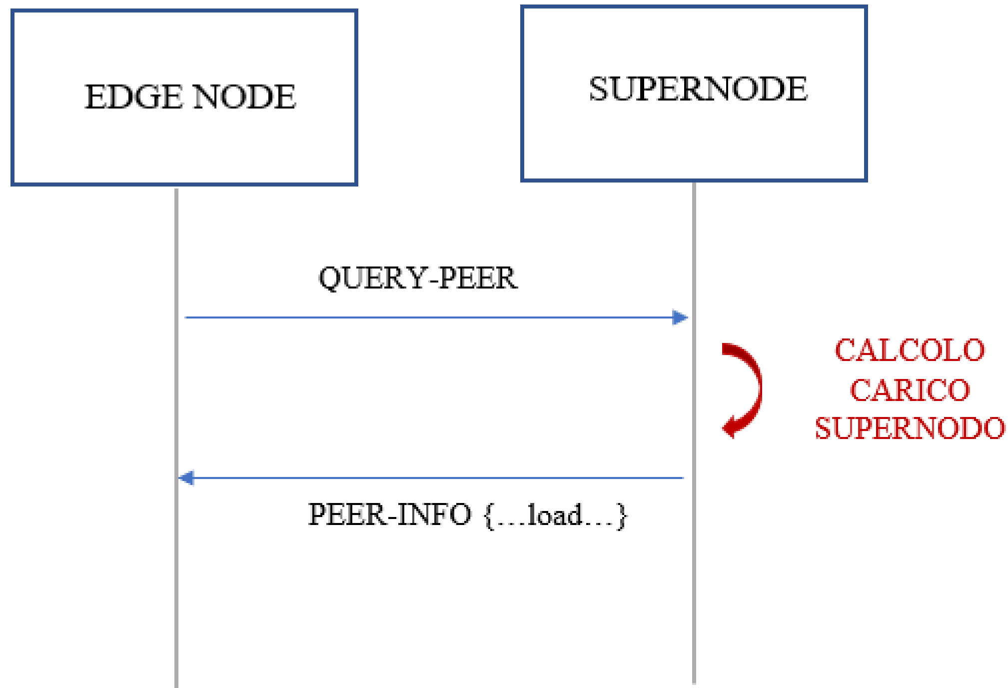

Figura 39: Invio di un messaggio QUERY_PEER e ricezione di un messaggio PEER_INFO nella strategia basata sul carico di lavoro di un supernodo.

La scelta riguardo quale tra le strategie di selezione analizzate implementare non è stata presa basandosi sulle loro performance, ma piuttosto sulla tipologia di informazione trasmessa al nodo edge. Infatti, alla base della strategia di selezione c'è l'informazione, la quale deve essere il più possibile informativa riguardo il supernodo per consentire un ordinamento, e successivamente una scelta da parte dell'edge, che rispecchi il più fedelmente possibile lo stato della rete.

Infatti, una strategia basata su RTT è semplice e di facile implementazione, ma non fornisce in maniera chiara e precisa la quantità di carico di lavoro di un supernodo. Come detto in precedenza, utilizzando una strategia di questo tipo, si rischierebbe di indirizzare gli edge come scelta verso quei supernodi più vicini ma anche più carichi.

Per questo motivo è stato scelto di implementare una strategia di selezione basata sul carico di lavoro di un supernodo. Essa infatti dispone i supernodi della lista in ordine crescente in base al valore del carico, in modo tale che il primo supernodo dopo l'ordinamento sia quello con minor carico di lavoro. Così facendo, quest'ultimo sarà distribuito nella maniera più equa possibile tra i supernodi attivi della rete.

Per verificare il corretto funzionamento della strategia e validare la scelta fatta, sono stati eseguiti diversi test utilizzando uno scenario con alcuni supernodi e diversi nodi edge registrati presso uno di essi. Dopodiché, è stato terminato il supernodo presso il quale erano registrati i nodi edge attraverso la combinazione di tasti Ctrl + C. Il corretto funzionamento della strategia di selezione è stato appurato eseguendo il comando *netcat -u localhost mgmt_port* sui supernodi e sui nodi edge: l'output ottenuto mostrava che i nodi edge si erano registrati presso altri supernodi in accordo al loro carico di lavoro.

Test simili sono stati eseguiti anche con la strategia basata su RTT: i risultati ottenuti sono stati in linea con quanto atteso, poiché il meccanismo di tale strategia e della registrazione presso altri supernodi era stato reso funzionante. Anche per quanto riguarda le performance, non sono state notate differenze significative e ciò avvalora ancora di più il fatto riportato in precedenza, per cui la scelta della strategia da implementare si è basata sul tipo di informazione trasmessa piuttosto che sulle prestazioni offerte.

Capitolo 5

Validazione

In questo capitolo sono presentati i test effettuati. Si tratta soprattutto di stress test, ossia esecuzioni del software sotto condizioni anomale, per verificare come il programma sia in grado di gestire tali eventi inattesi e quali sono le performance in queste situazioni. Nel seguito del capitolo sono analizzate due situazioni anomale, riconducibili rispettivamente ad un attacco esterno ed un attacco interno alla rete n2n.

Infine, sono riportati i risultati finali come validazione del software con le funzionalità introdotte e le modifiche apportate. Per validazione si intende un controllo mirato a confrontare i risultati ottenuti con gli obiettivi preposti, i requisiti iniziali e/o con altri software equivalenti [4].

5.1 Attacchi esterni

Un attacco esterno è tipicamente condotto da uno o più punti al di fuori della rete e tenta di interrompere il servizio. Gli unici punti critici in una comunicazione n2n vulnerabili dall'esterno sono i messaggi PING e REGISTER, poiché chiunque si presenti con una comunità apparentemente valida sarà in grado di inviare richieste di questo tipo.

Questa è stata una delle ragioni per cui su n2n è stata sviluppata la crittografia dell'intestazione, poiché solo il nome della comunità (da mantenere segreto, non appare come testo in chiaro nei messaggi) consente la comunicazione con il resto della rete. Utilizzando la crittografia dell'intestazione, anche la comunicazione interna della federazione è protetta e accessibile solo per i supernodi della stessa federazione, il cui nome funge da chiave. Infatti, solo la conoscenza di quest'ultima consentirebbe di disturbare quella comunicazione.

Tuttavia, il supernodo non ha protezione contro gli attacchi di flooding [10]. Assumendo di avere la cifratura dell'intestazione abilitata, questo genere di attacco, in teoria, comporterebbe il controllo della decifrazione dell'intestazione per ogni singolo pacchetto in arrivo, trascinando la velocità di elaborazione del supernodo verso il basso.

Si è deciso di testare questo fenomeno utilizzando un breve programma C, ottenuto prendendo un esempio di Client UDP [13] modificato ad hoc, che si connette alla porta UDP di un supernodo e invia ripetutamente pacchetti casuali di lunghezza maggiore di 20 bytes. Se i pacchetti fossero inferiori a 20 bytes, il supernodo li scarterà automaticamente.

```c
int main() {
    int sockfd;
    char *flood = "floodMessageToTheSupernode";
    struct sockaddr_in     servaddr;

    // Creating socket file descriptor
    if((sockfd = socket(AF_INET, SOCK_DGRAM, 0)) < 0) {
        perror("socket creation failed");
        exit(EXIT_FAILURE);
    }

    memset(&servaddr, 0, sizeof(servaddr));

    // Filling server information
    servaddr.sin_family = AF_INET;
    servaddr.sin_port = htons(PORT);
    servaddr.sin_addr.s_addr = inet_addr(ADDRESS);

    int n, len;

    while(1) {
        sendto(sockfd, (const char *)flood, strlen(flood),
               MSG_CONFIRM, (const struct sockaddr *) &servaddr,
               sizeof(servaddr));
    }
}
```

Figura 40: Corpo del client UDP utilizzato per realizzare il test.

Il programma in Figura 40 crea quindi un socket, lo inizializza con l'indirizzo IP e la porta del supernodo vittima e, in un ciclo teoricamente infinito, invia continuamente un pacchetto UDP generato casualmente. Lanciando un'istanza di un supernodo con la sua porta UDP, abilitando più volte la modalità verbose - ad esempio supernode -p 3001 -vvvvvvvvvvvvvv - ed eseguendo una o più istanze del client UDP, il risultato ottenuto sarà dunque come quello mostrato, in parte, nella Figura 41.

```
[sn.c:621] traceLevel is 15
[sn.c:628] supernode is listening on UDP 3001 (main)
[sn.c:636] supernode is listening on UDP 5645 (management)
[sn.c:644] Dropping privileges to uid=65534, gid=65534
[sn.c:659] supernode started
[sn_utils.c:805] Processing incoming UDP packet [len: 26][sender: 192.168.1.227:51041]
[sn_utils.c:874] process_udp dropped a packet with seemingly encrypted header for which no matching community which uses encrypted headers was found.
[sn_utils.c:627] Purging old communities and edges
[sn_utils.c:642] Remove 0 edges
[sn_utils.c:805] Processing incoming UDP packet [len: 26][sender: 192.168.1.227:51041]
[sn_utils.c:874] process_udp dropped a packet with seemingly encrypted header for which no matching community which uses encrypted headers was found.
[sn_utils.c:805] Processing incoming UDP packet [len: 26][sender: 192.168.1.227:51041]
[sn_utils.c:874] process_udp dropped a packet with seemingly encrypted header for which no matching community which uses encrypted headers was found.
[sn_utils.c:805] Processing incoming UDP packet [len: 26][sender: 192.168.1.227:51041]
[sn_utils.c:874] process_udp dropped a packet with seemingly encrypted header for which no matching community which uses encrypted headers was found.
[sn_utils.c:805] Processing incoming UDP packet [len: 26][sender: 192.168.1.227:51041]
[sn_utils.c:874] process_udp dropped a packet with seemingly encrypted header for which no matching community which uses encrypted headers was found.
[sn_utils.c:805] Processing incoming UDP packet [len: 26][sender: 192.168.1.227:51041]
[sn_utils.c:874] process_udp dropped a packet with seemingly encrypted header for which no matching community which uses encrypted headers was found.
```

Figura 41: Parte dell'output ottenuto eseguendo un'istanza di supernodo (supernode -p 3001) e un'istanza di client UDP (./udp_tool).

Da tale figura si può notare che il client UDP, con indirizzo 192.168.1.227 e porta 51041, invia ripetutamente pacchetti UDP al supernodo sulla sua porta UDP 3001. Il loro invio continuerà fintanto che il client non verrà terminato. Inoltre, ciò che si può vedere, è che il supernodo scarta i pacchetti ricevuti fornendo la seguente motivazione: la funzione process_udp "ha scartato un pacchetto con un'intestazione apparentemente crittografata per la quale non è stata trovata alcuna comunità corrispondente che utilizza intestazioni crittografate".

Andando a ispezionare la porzione di codice che genera questo messaggio, si può vedere che in quel punto c'è un apposito controllo per assicurarsi che nessun pacchetto non crittografato possa essere iniettato in una comunità con intestazioni crittografate. Inoltre, se i pacchetti inviati dal client UDP fossero crittografati, la funzione *packet_header_decryption* nel file src/header_encryption.c gestirebbe questa problematica nel modo corretto.

Infatti, analizzando il corpo di questa funzione, si potrà vedere che un pacchetto cifrato viene identificato controllando solamente pochi bytes iniziali e il processo di decifrazione continuerà solamente se si trovano i bytes corrispondenti al magic number (come spiegato nel Capitolo 3 nel paragrafo riguardante la crittografia in n2n.). In questo modo viene fatto solo un piccolo sforzo, e non avviene una decifrazione completa dell'intestazione di un pacchetto per controllare la sua validità. Quindi, non ci vorrà troppa CPU per verificare e scartare i pacchetti generati casualmente da un attacco di flooding, come quello portato avanti dall'esecuzione del client UDP.

Il supernodo riuscirà a scartare i pacchetti casuali ricevuti perché n2n utilizza un cifrario ARX (Addition, Rotation and XOR) veloce, ossia SPECK [1], per la cifratura e la decifrazione dell'intestazione, e quindi potrebbe persino accadere che riesca a mantenere il passo con pacchetti in arrivo alla massima velocità di collegamento. Si può quindi affermare che un qualsiasi pacchetto UDP di almeno 20 bytes richiederà solo una piccola quantità di lavoro per essere decifrato.

Per quanto riguarda la resistenza ad un attacco di flooding, sono stati eseguiti dei test utilizzando due supernodi appartenenti alla federazione, due nodi edge registrati presso un singolo supernodo e istanze multiple del client UDP. Il supernodo presso il quale entrambi gli edge si sono registrati è stato l'obiettivo dell'attacco. In fase di avvio, prima di eseguire i client UDP, è rimasto tutto invariato: i due supernodi si sono registrati l'un l'altro e i nodi edge si sono registrati presso il supernodo specificato con l'opzione "-l". Verificando la situazione tramite il comando *netcat* sulla porta UDP di management del supernodo vittima, si è ottenuto l'output riportato in Figura 42.

```
   id    tun_tap            MAC                edge                 hint             last_seen
---------------------------------------------------------------------------------------------------
community: *Federation
   1     0.0.0.0/0          72:82:9E:45:33:15  192.168.1.227:3000                    10
community: XXXXXX
   1     192.168.8.1/24     4A:30:BF:02:BF:DB  192.168.1.227:7000   francesco-Virtu  28
   2     192.168.8.2/24     F6:E6:C0:81:6F:84  192.168.1.40:7001    francesco-Virtu  20
---------------------------------------------------------------------------------------------------
```

Figura 42: Output ottenuto dalla porta UDP di management del supernodo vittima, eseguendo il comando netcat -u localhost 3000.

Dopo un certo periodo di tempo sono state avviate le istanze dei client UDP, le quali hanno iniziato a generare una quantità elevata di traffico verso il supernodo vittima. Andando a controllare l'output fornito da esso, è stato notato che tutti questi pacchetti venivano scartati, come in Figura 41. La particolarità che però è stato possibile osservare riguarda il fatto che i due nodi edge erano comunque in grado di comunicare: infatti, anche provando un semplice comando di ping, si poteva vedere che essi riuscivano a raggiungersi l'un l'altro. Ciò è stato reso possibile dalle funzionalità introdotte durante questo lavoro, in particolare grazie alla comunicazione tra supernodi. Infatti, andando nuovamente ad eseguire il comando netcat sui due supernodi, sia su quello vittima sia sull'altro, si è potuto osservare una situazione ben chiara.

I nodi edge erano stati rimossi dal supernodo vittima poiché essi erano impossibilitati a contattarlo a causa dell'elevata quantità di traffico in entrata su quel supernodo. Ma essi, grazie alla comunicazione tra supernodi e alle informazioni propagate all'interno della rete, nel frattempo erano venuti a conoscenza dell'esistenza di un altro supernodo e si erano registrati presso di lui. Quest'ultimo passaggio è stato reso possibile anche grazie alla strategia di selezione di un supernodo basata sul carico: infatti, andando ad eseguire netcat sulla porta di management di uno dei due edge, si potrà constatare che il supernodo sotto attacco è l'ultimo della lista di supernodi, e quindi non riceverà richieste di registrazione. Infine, eseguendo il comando netcat su entrambi i supernodi, si è potuto ottenere l'output mostrato in Figura 43 e Figura 44.

```
   id    tun_tap            MAC                edge                   hint            last_seen
-----------------------------------------------------------------------------------------------
community: *Federation
    1    0.0.0.0/0          72:82:9E:45:33:15  192.168.1.227:3000                     41
-----------------------------------------------------------------------------------------------
```

Figura 43: Output ottenuto dalla porta UDP di management del supernodo vittima durante l'attacco, eseguendo il comando netcat -u localhost 3000.

```
   id    tun_tap             MAC                edge                   hint             last_seen
------------------------------------------------------------------------------------------------
community: *Federation
   1     0.0.0.0/0           CA:71:E6:33:0A:5B  192.168.1.40:3001                       19
community: XXXXXX
   1     192.168.8.1/24      4A:30:BF:02:BF:DB  192.168.1.227:7000     francesco-Virtu  11
   2     192.168.8.2/24      F6:E6:C0:81:6F:84  192.168.1.40:7001      francesco-Virtu  2
------------------------------------------------------------------------------------------------
```

Figura 44: Output ottenuto dalla porta UDP di management dell'altro supernodo durante l'attacco, eseguendo il comando netcat -u localhost 3001.

Si è dunque potuto comprendere bene che i due nodi edge, grazie alle funzionalità introdotte, come la comunicazione tra supernodi e la strategia di selezione di un supernodo basata sul carico, sono riusciti ad elaborare le informazioni della rete e a sfruttarle a proprio vantaggio per registrarsi presso un altro supernodo esistente e continuare le proprie operazioni.

Introducendo delle modifiche allo strumento benchmark, è stato possibile conoscere il numero di pacchetti cifrati che la CPU di un computer è in grado di gestire. L'output ottenuto dal benchmark fornisce il numero di pacchetti al secondo che un supernodo o un edge è in grado di identificare come errati e che verranno scartati. Questo valore andrà diviso per il numero di comunità con crittografia dell'header abilitata. Sicuramente di comunità con questa caratteristica si ha la federazione dei supernodi, dopodiché si potrà assumere dalle 2 alle 3 comunità regolari, quindi un supernodo gestisce in genere 4 comunità con crittografia dell'intestazione abilitata. Questo avviene perché per ogni pacchetto cifrato in arrivo, il supernodo controllerà 4 comunità. Eseguendo questo strumento sui due PC utilizzati per fare il test illustrato sopra, si è ottenuto i seguenti risultati:

- *PC1*: 11.125 pps / 4 = 2.781 pps. Dato che un singolo pacchetto avrà una dimensione di almeno 20 bytes, un attaccante avrà bisogno approssimativamente di un bitrate di 450 Mb/s, o equivalentemente di 56 MB/s, per saturare un supernodo su questo PC e renderlo irraggiungibile.

- *PC2*: 5.638 pps / 4 = 1.409 pps. Dato che un singolo pacchetto avrà una dimensione di almeno 20 bytes, un attaccante avrà bisogno approssimativamente di un bitrate di 225 Mb/s, o equivalentemente di 28 MB/s, per saturare un supernodo su questo PC e renderlo irraggiungibile.

Lo strumento di benchmark è ottimo per mostrare anche quale differenza può fare utilizzare flag diversi durante la compilazione del programma. L'output analizzato in precedenza differisce molto tra ./configure CFLAGS = ”” e ./configure CFLAGS = ”-O3 -march = native”. Andando ad eseguire nuovamente benchmark sul PC2 con i flag di compilazione abilitati, si è ottenuto come output un valore di 12.231 pps. Facendo le stesse considerazioni riportate in precedenza, in questo caso un attaccante avrà bisogno approssimativamente di un bitrate di 990 Mb/s, o equivalentemente di 124 MB/s, per saturare un supernodo in esecuzione su PC2.

5.2 Attacchi interni

Un attacco interno alla rete n2n consiste in una situazione in cui un nodo edge o un supernodo si comportano in modo anomalo e diventano una minaccia per la disponibilità della rete stessa. Sono quindi state analizzate due particolari situazioni di attacchi interni ed è stato esaminato come le entità di n2n reagiscono a situazioni di questo tipo.
Gli scenari considerati riguardano l'utilizzo, sia da parte di edge che di supernodi, di indirizzi MAC già in uso oppure generati casualmente in quantità elevate e diversi ogni volta.

5.2.1 Indirizzi MAC duplicati

Per quanto riguarda l'utilizzo di indirizzi MAC già in uso, eseguendo un semplice test con un supernodo e due nodi edge con stesso MAC, è stato possibile osservare che i due nodi edge si sovrascrivevano presso il supernodo e non erano in grado di comunicare. Una situazione di questo tipo è potenzialmente pericolosa, perché un attaccante potrebbe conoscere l'indirizzo MAC di un dato nodo edge e connettersi volutamente alla rete con quello stesso indirizzo per impedirgli la comunicazione. Per risolvere questo problema è stato deciso di implementare un apposito meccanismo di protezione contro l'utilizzo di MAC duplicati che si basa su un nuovo tipo di messaggio, chiamato UNREGISTER_SUPER, e su un campo *auth* contenuto nella struttura di un messaggio REGISTER_SUPER e finora inutilizzato.

Lo schema di autenticazione implementato è basato su un numero di identificazione univoco, il quale viene generato in modo casuale durante l'avvio di un nodo edge e rimane invariato fino a che esso non termina. Ad ogni messaggio REGISTER_SUPER, il token di autenticazione viene trasmesso al supernodo, il quale lo ha memorizzato dal primo contatto avvenuto tra le due entità, e potrà quindi confrontarlo. Il supernodo accetterà messaggi REGISTER_SUPER e UNREGISTER_SUPER solo se la verifica dell'identificativo avrà esito positivo.

Questo impedisce l'utilizzo di MAC duplicati anche in caso di più supernodi presenti nella federazione, perché ogni messaggio REGISTER_SUPER viene inoltrato a tutti gli altri supernodi. Se un nuovo nodo edge tenta intenzionalmente, o accidentalmente, di richiedere un indirizzo MAC già in uso, ciò verrebbe rilevato come modifica non autorizzata poiché il nuovo edge presenta un identificativo diverso da quello memorizzato. Il supernodo che rileva questa anomalia invierà un messaggio REGISTER_SUPER_NAK (Figura 45) al nodo edge ed anche al supernodo presso il quale esso ha tentato di registrarsi. Il messaggio REGISTER_SUPER_NAK farà arrestare il programma del nodo edge, informando l'utente, e causerà la cancellazione delle sue informazioni presso il supernodo.

```
typedef struct n2n_REGISTER_SUPER_NAK
{
  n2n_cookie_t        cookie;
  n2n_mac_t           srcMac;
} n2n_REGISTER_SUPER_NAK_t;
```

Figura 45: Struttura di un messaggio REGISTER_SUPER_NAK.

Nel momento in cui il nodo edge riceve il pacchetto REGISTER_SUPER_NAK, dopo averlo decodificato, controlla che il campo *srcMac* coincida con il proprio indirizzo MAC. Se i due indirizzi MAC sono uguali, allora il nodo edge termina con un messaggio di errore per avvisare l'utente, altrimenti cerca il nodo edge con MAC duplicato o falsificato all'interno delle liste pending_peers e known_peers e, se lo trova, lo rimuove.

A differenza dell'indirizzo MAC, che viene scambiato in qualsiasi tipo di messaggio, anche direttamente tra nodi edge, l'identificativo memorizzato nel campo *auth* di un nodo edge sarà noto solo al supernodo. Altri nodi edge non potranno conoscerlo e quindi non saranno in grado di falsificarlo.

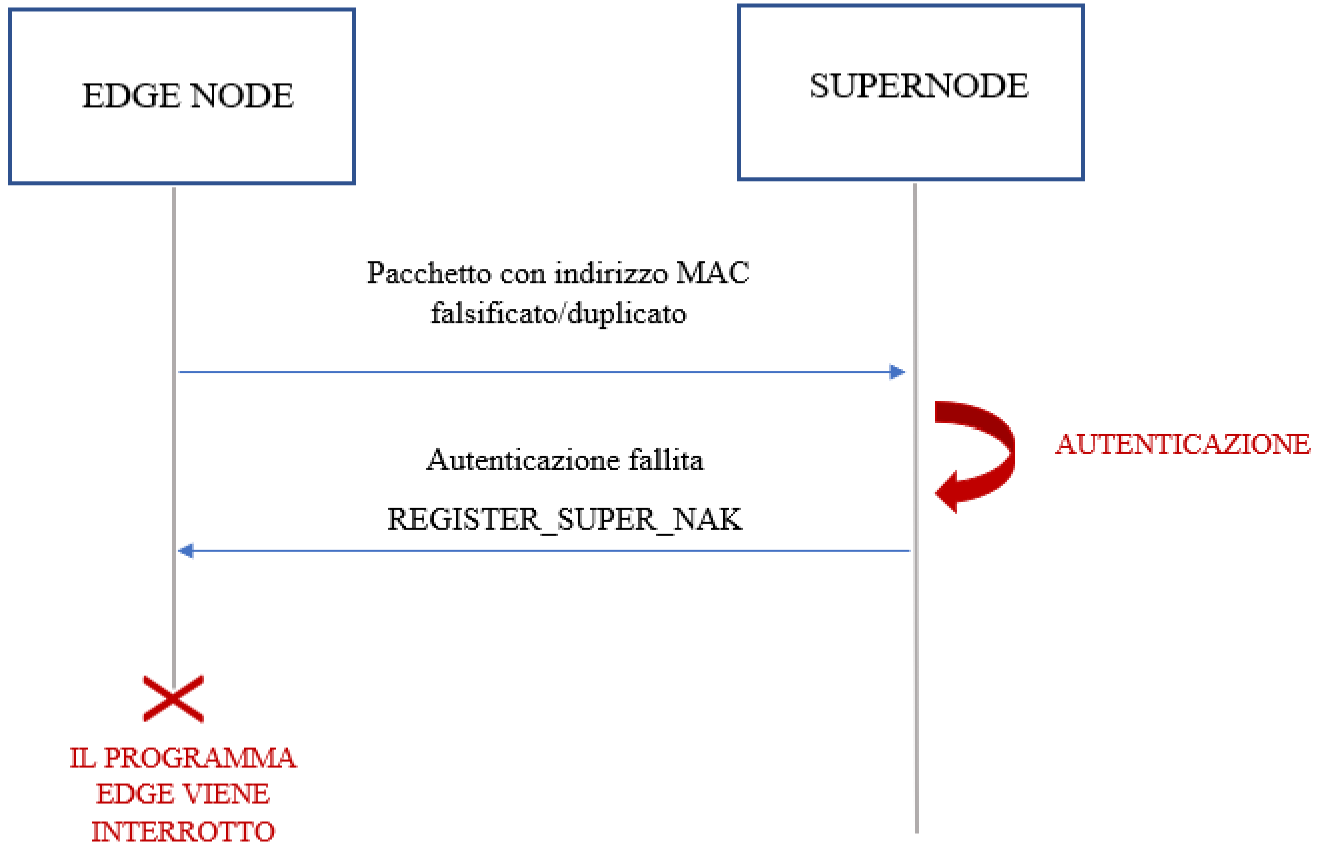

Figura 46: Invio di un messaggio REGISTER_SUPER con MAC falsificato/duplicato e ricezione del messaggio REGISTER_SUPER_NAK.

Il nuovo pacchetto UNREGISTER_SUPER (Figura 47) viene invece utilizzato quando un nodo edge cambia il supernodo presso il quale è registrato o quando termina: in questo modo esso non lascerà informazioni inconsistenti di sé stesso nelle strutture dati del supernodo al quale era registrato in precedenza.

```
typedef struct n2n_UNREGISTER_SUPER
{
  n2n_auth_t auth;
  n2n_mac_t srcMac;
} n2n_UNREGISTER_SUPER_t;
```

Figura 47: Struttura di un messaggio UNREGISTER_SUPER.

Anche esso ha un campo *auth* poiché potrebbe avvenire un attacco nel quale viene inviato un messaggio UNREGISTER_SUPER con indirizzo MAC della vittima in modo tale da farla eliminare dalle liste di edge di un supernodo. Se anche un messaggio di questo tipo non fosse autenticato, la situazione appena descritta potrebbe impedire al nodo edge vittima di continuare a svolgere le proprie operazioni. Per questo, tramite lo stesso meccanismo di autenticazione descritto in precedenza, un supernodo verifica l'autenticità anche di questo ulteriore messaggio. Se ciò va a buon fine, il nodo edge si disconnette dalla rete e le sue informazioni verranno eliminate dal supernodo.

La debolezza dello schema di protezione descritto in precedenza è data dal fatto che un attacco di sniffing condotto sapientemente, e volto a conoscere l'identificativo, potrebbe romperlo. Pertanto, come riportato anche nel Capitolo 7, è previsto un ulteriore sviluppo verso uno schema di autenticazione basato su crittografia più sofisticato e più difficilmente attaccabile.

5.2.2 Indirizzi MAC falsificati

Un attacco di MAC spoofing è un attacco in cui un nodo edge invia numerosi pacchetti all'interno della rete, ognuno dei quali avrà un indirizzo MAC diverso calcolato casualmente. La minaccia rilevata a seguito di questo attacco consiste nella crescita a dismisura delle liste contenenti i nodi edge, le quali si basano appunto sugli indirizzi MAC utilizzati come chiave. Una struttura dati che cresce a dismisura nel tempo è una minaccia per qualsiasi tipo di software, poiché diventa presto ingestibile, occupa molta memoria e qualsiasi tipo di operazione ha un costo elevato.

I supernodi erano già in grado di gestire questa situazione, grazie alla ricerca addizionale basata sull'interfaccia TAP all'interno della funzione *update_edge* e al meccanismo di autenticazione illustrato in precedenza. La ricerca basata su TAP, già presente nel codice, consente di trovare un nodo edge anche se il suo MAC è stato alterato e permette quindi di evitare l'aggiunta di nuovi edge.

Per questi ultimi però il discorso è differente. Infatti, provando ad alterare gli indirizzi MAC durante l'invio di messaggi REGISTER, è stato constatato, tramite l'output della porta di management, che le liste di nodi edge tendevano a crescere sempre di più. Quindi è stato implementato un meccanismo di protezione basato su una ricerca aggiuntiva per socket, da eseguire all'interno delle liste known_peers e pending_peers di un nodo edge (Figura 48).

```
static struct peer_info* find_peer_by_sock(const n2n_sock_t *sock, struct peer_info *peer_list){
  struct peer_info *scan, *tmp, *ret = NULL;

  HASH_ITER(hh, peer_list, scan, tmp){
    if(memcmp(&(scan->sock), sock, sizeof(n2n_sock_t)) == 0){
      ret = scan;
      break;
    }
  }

  return ret;
}
```

Figura 48: Funzione che cerca all'interno della lista, passata come parametro, l'eventuale presenza di un elemento con il socket di interesse.

La precedente funzione, se lo trova, restituirà un nodo edge il cui socket è uguale a quello cercato. Essa è invocata nelle parti di codice di src/edge_utils.c dove si utilizzano le macro HASH_FIND_PEER e HASH_ADD_PEER. Infatti, la macro HASH_FIND_PEER utilizza il MAC come chiave di ricerca, quindi in caso di un attacco di MAC spoofing potrebbe non essere in grado di trovare il nodo cercato. Ma, chiamando subito dopo la nuova funzione mostrata sopra, verrà effettuata una seconda ricerca per socket, per essere sicuri che il nodo in questione non sia veramente già presente nella rete. Così facendo, si eviterà di aggiungere un ulteriore nodo edge alla lista ed in questo modo essa non crescerà più a dismisura.

5.3 Risultati

Gli obiettivi fissati e le funzionalità implementate per raggiungerli sono stati riassunti brevemente nella seguente tabella.

Obiettivo	**Funzionalità**	**Risultato raggiunto**
Comunicazione tra supernodi	Federazione di supernodi	SI
Propagazione stato della rete	Payload	SI
Strategia di selezione su nodi edge	Carico supernodo	SI
Resistenza ad attacchi interni	Autenticazione con token	SI

L'esecuzione di test in condizioni normali, ma soprattutto di stress test come quelli illustrati in precedenza, ha aiutato a comprendere come le nuove funzionalità si siano inserite bene con quelle già esistenti. In questo modo, sono stati raggiunti tutti gli obiettivi posti in partenza, ma soprattutto sono stati rispettati i meccanismi già esistenti e funzionanti delle precedenti versioni di n2n.

L'ampia fase di studio del codice esistente, e il confronto quasi quotidiano con gli altri sviluppatori di n2n, tramite Github e canale Discord, sono state fasi fondamentali per la buona riuscita di questo progetto di tirocinio e per il raggiungimento degli obiettivi preposti.

Capitolo 6

Conclusione

Come già discusso, l'ampia diffusione della tecnologia IoT e l'avvento ormai prossimo del 5G accresceranno sempre più la quantità di dispositivi connessi in una rete, ma anche la potenza e la velocità di trasmissione della stessa. Si rende quindi necessario ricorrere sempre più alla tecnologia peer-to-peer in modo che le comunicazioni di una rete e le informazioni processate al suo interno non siano dipendenti da poche entità centralizzate e che queste comunicazioni siano totalmente sicure.

Il progetto portato avanti durante questo tirocinio formativo ha avuto come obiettivo quello di prendere un software VPN esistente, come n2n, e ridefinire il suo protocollo di comunicazione per renderlo totalmente distribuito, come dimostrato dalle funzionalità introdotte ed illustrate nel Capitolo 4.

Nello specifico, sono state approfonditi come prima cosa due argomenti alla base delle reti virtuali distribuite, ossia la tecnologia peer-to-peer e le DHTs. Dopodiché, si è resa necessaria una fase di studio approfondito del codice e di analisi dettagliata delle funzionalità già esistenti, per comprendere a fondo il comportamento del software n2n. A ciò è seguita una fase di confronto quasi costante con gli altri sviluppatori di n2n, sia italiani che stranieri, tramite GitHub e canale Discord, per concordare quali fossero le nuove funzionalità da implementare e in che modo operare. Questa fase in particolare è risultata molto importante perché ha consentito discussioni sulle nuove funzionalità

da aggiungere e ha permesso di fissare in modo chiaro e definito quali fossero i punti critici, sia in termine di codice che di funzioni, da non alterare per non compromettere il corretto funzionamento del software esistente. Una volta fissati questi nuovi meccanismi, sono state apportate al codice di n2n le modifiche necessarie per raggiungere gli obiettivi di partenza.

Infine, nel Capitolo 5, sono stati riportati i risultati ottenuti in rapporto ai vari obiettivi intermedi preposti. Essi, facenti parte dell'obiettivo primario e generale di estensione del protocollo di comunicazione, sono stati raggiunti con successo, anche grazie alle nuove funzionalità implementate. Tramite esse, il protocollo di comunicazione di n2n è stato reso totalmente distribuito, consentendo al software di sfruttare le caratteristiche e lo stato della rete per poter effettuare le operazioni al meglio. Inoltre, sono stati illustrati i meccanismi tramite i quali il software n2n è in grado di resistere ad alcune situazioni anomale, come ad esempio attacchi di sicurezza, sia esterni che interni.

Capitolo 7

Lavori futuri

In futuro potrebbero essere apportate ulteriori migliorie al software n2n. Ad esempio potrebbe essere utile verificare periodicamente il contenuto della lista di supernodi presso i nodi edge. In questo caso sarà necessario adottare una strategia di eliminazione probabilmente simile a quella utilizzata lato supernodo con la funzione re_register_and_purge_supernodes. Il codice esistente potrà essere riutilizzato, definendo una funzione comune all'interno del file src/n2n.c, prestando però attenzione a dividere in maniera efficiente i meccanismi di re-registrazione ed eliminazione.

Un'altra piccola modifica che potrebbe essere introdotta lato edge riguarda l'utilizzo di timestamp più precisi, ad esempio quelli utilizzati per la crittografia dei pacchetti. In questo modo si aumenterebbe l'accuratezza, passando da secondi a millisecondi, apportando evidenti benefici e maggior precisione a quelle porzioni di codice direttamente coinvolte.

Parlando poi di strategie di selezione, le due analizzate nel Capitolo 4 sono di semplice comprensione ed implementazione, ma allo stesso tempo molto basilari. Perciò un'ulteriore miglioria potrebbe consistere nel cercare una strategia di selezione ibrida, che tenga conto di più fattori assieme (ad esempio RTT, carico di lavoro, locazione fisica di un supernodo, ecc.) per consentire ad un nodo edge la miglior scelta possibile basandosi su più informazioni che lo aiutino a selezionare il supernodo più adatto.

Il punto critico riguardo la strategia di selezione attualmente implementata è dato dal fatto che, per il momento, l'utente non potrà utilizzarne di altre o scegliere quale adottare, ma dovrà necessariamente attenersi alla sola strategia implementata in quel preciso momento. Questo perché il passaggio da un strategia ad un'altra richiede un cambiamento di approccio completo. Perciò, una modifica da introdurre potrebbe essere quella di offrire meccanismi per mantenere diverse strategie in parallelo e passare, in maniera efficiente, da una all'altra. Ciò includerebbe un altro livello di astrazione come quello usato per le trasformazioni crittografiche, in cui ognuna segue uno schema univoco e la selezione viene eseguita utilizzando i puntatori a funzione. Sarebbe quindi una buona scelta quella di introdurre un'opzione dalla riga di comando che consenta all'utente, in maniera facoltativa, di poter selezionare una strategia tra quelle implementate.

Un'altra miglioria apportabile al codice, come già detto nel Capitolo 5, riguarda il meccanismo di autenticazione degli edge. Lo schema implementato attualmente è funzionante ma certamente piuttosto semplice, tanto che un attacco di sniffing avanzato potrebbe romperlo. Quindi l'implementazione di un meccanismo di autenticazione, realizzata in questo lavoro, può essere vista come un primo passo verso schemi più sofisticati, basati sulla crittografia, che potranno essere sviluppati in futuro.

Infine, sarebbe molto importante, in particolare per i nodi edge, sviluppare una dashboard o un'interfaccia utente, utilizzabile anche da sistemi operativi come Windows e MacOS. In questo modo, tramite un'interfaccia grafica di questo tipo, sarebbe possibile per un utente avere informazioni più dettagliate e più facilmente visibili sulla propria comunità e sui membri di quest'ultima.

Capitolo 8

Appendice

8.1 Codice sorgente

Il codice sorgente ed il file README.md che mostra i requisiti, la compilazione e l'esecuzione del software, è reperibile al seguente link:
`https://github.com/ntop/n2n`

Riferimenti

[1] Ray Beaulieu et al. “SIMON and SPECK: Block Ciphers for the Internet of Things.” In: *IACR Cryptol. ePrint Arch.* 2015 (2015), p. 585.

[2] Anna Bernasconi, Paolo Ferragina e Fabrizio Luccio. *Elementi di crittografia.* Pisa University Press, 2015.

[3] Daniel J Bernstein. “ChaCha, a variant of Salsa20”. In: *Workshop Record of SASC*. Vol. 8. 2008, pp. 3–5.

[4] Giovanni A Cignoni, Carlo Montangero e Laura Semini. “Il controllo del software: verifica e validazione”. In: (2009).

[5] Alem Čolaković e Mesud Hadžialić. “Internet of Things (IoT): A review of enabling technologies, challenges, and open research issues”. In: *Computer Networks* 144 (2018), pp. 17–39.

[6] Coursera. *Distributed Hash Tables.* URL: `https://www.coursera.org/lecture/data-structures/distributed-hash-tables-tvH8H`.

[7] Luca Deri e Richard Andrews. “N2n: A layer two peer-to-peer vpn”. In: *IFIP International Conference on Autonomous Infrastructure, Management and Security*. Springer. 2008, pp. 53–64.

[8] M. Dufel. *Distributed Hash Tables And Why They Are Better than Blockchain For Exchanging Health Records.* 2017.

[9] Edpresso. *What is a distributed hash table.* URL: `https://www.educative.io/edpresso/what-is-a-distributed-hash-table`.

[10] eSecurityPlanet. *Types of DDoS Attacks*. URL: https://www.esecurityplanet.com/network-security/types-of-ddos-attacks.html.

[11] Fastweb. *Peer-to-peer network*. URL: https://www.fastweb.it/internet/cosa-e-come-funziona-p2p/.

[12] FreePN. *FreePN*. URL: https://www.freepn.org/.

[13] GeeksforGeeks. *UDP Server-Client implementation in C*. URL: https://www.geeksforgeeks.org/udp-server-client-implementation-c/.

[14] T.D. Hanson e A. O'Dwyer. *uthash User Guide*. URL: https://troydhanson.github.io/uthash/userguide.html.

[15] M Frans Kaashoek e David R Karger. "Koorde: A simple degree-optimal distributed hash table". In: *International Workshop on Peer-to-Peer Systems*. Springer. 2003, pp. 98–107.

[16] M. Krasnyansky. *Universal TUN/TAP Driver*. URL: http://vtun.sourceforge.net/.

[17] Eng Keong Lua et al. "A survey and comparison of peer-to-peer overlay network schemes". In: *IEEE Communications Surveys & Tutorials* 7.2 (2005), pp. 72–93.

[18] Petar Maymounkov e David Mazieres. "Kademlia: A peer-to-peer information system based on the xor metric". In: *International Workshop on Peer-to-Peer Systems*. Springer. 2002, pp. 53–65.

[19] ntop. *n2n - A Layer Two Peer-to-Peer VPN*. URL: https://www.ntop.org/products/n2n/.

[20] ntop. *n2n: Peer-to-peer VPN*. URL: https://github.com/ntop/n2n.

[21] M. Oberhumer. *LZO real-time data compression library*. URL: http://www.oberhumer.com/opensource/lzo/.

[22] David C Plummer et al. "Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48. bit Ethernet address for transmission on Ethernet hardware." In: *RFC* 826 (1982), pp. 1–10.

[23] Mirza Abdur Razzaq et al. "Security issues in the Internet of Things (IoT): a comprehensive study". In: *International Journal of Advanced Computer Science and Applications* 8.6 (2017), p. 383.

[24] Bruce Schneier et al. *The Twofish encryption algorithm: a 128-bit block cipher*. John Wiley & Sons, Inc., 1999.

[25] Rüdiger Schollmeier. "A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications". In: *Proceedings First International Conference on Peer-to-Peer Computing*. IEEE. 2001, pp. 101–102.

[26] Ion Stoica et al. "Chord: a scalable peer-to-peer lookup protocol for internet applications". In: *IEEE/ACM Transactions on networking* 11.1 (2003), pp. 17–32.

[27] Tailscale. *Tailscale*. URL: `https://tailscale.com`.

[28] Toptal. *Consistent Hashing*. URL: `https://www.toptal.com/big-data/consistent-hashing`.

[29] Wikipedia. *Chord*. URL: `https://en.wikipedia.org/wiki/Chord`.

[30] Wikipedia. *Cryptographic nonce*. URL: `https://en.wikipedia.org/wiki/Cryptographic_nonce`.

[31] Wikipedia. *Distributed hash table*. URL: `https://en.wikipedia.org/wiki/Distributed_hash_table`.

[32] Wikipedia. *Exclusive or*. URL: `https://en.wikipedia.org/wiki/Exclusive_or`.

[33] Wikipedia. *Kademlia*. URL: `https://en.wikipedia.org/wiki/Kademlia`.

[34] Wikipedia. *Koorde*. URL: `https://en.wikipedia.org/wiki/Koorde`.

[35] Wikipedia. *Multi-factor authentication*. URL: `https://en.wikipedia.org/wiki/Multi-factor_authentication`.

[36] Wikipedia. *Pearson hashing*. URL: `https://en.wikipedia.org/wiki/Pearson_hashing`.

[37] Wikipedia. *Tor (anonymity network)*. URL: https://en.wikipedia.org/wiki/Tor_(anonymity_network).

[38] ZeroTier. *ZeroTier*. URL: https://www.zerotier.com/.

[39] Haijun Zhang et al. "Network slicing based 5G and future mobile networks: mobility, resource management, and challenges". In: *IEEE communications magazine* 55.8 (2017), pp. 138–145.