Koorde DHT

Progetto B4: Small-scale DHT system

Flavio Simonelli
Corso di Sistemi distribuiti e Cloud Computing
Unversità degli Studi di Roma "Tor Vergata"
Matricola: 0365333
Flavio.simonelli@alumni.uniroma2.eu

***Abstract*—Questo elaborato presenta lo studio, la progettazione, l'implementazione e il test di una Distributed Hash Table (DHT) basata sull'algoritmo Koorde, una variante di *Chord* che sfrutta la struttura dei grafi di De Bruijn per ottimizzare il compromesso tra grado di connessione e numero di hop necessari per le operazioni di lookup. L'obiettivo del lavoro è analizzare i principi teorici che rendono Koorde una DHT degree-ottimale e realizzarne un'implementazione completa, sviluppata in linguaggio Go, configurabile e utilizzabile in contesti distribuiti. L'elaborato include inoltre la dimostrazione del deploy in ambiente cloud AWS, con l'integrazione di strumenti di monitoraggio e osservabilità per l'analisi delle prestazioni, della latenza media e della resilienza del sistema in condizioni di churn dinamico.**

I. Introduzione

Negli ultimi decenni, l'evoluzione delle architetture dei sistemi distribuiti ha portato a un progressivo spostamento da modelli **centralizzati**, basati su server dedicati, a soluzioni **decentralizzate**, più flessibili, scalabili e resilienti. In questo contesto, le **reti peer-to-peer (P2P)** rappresentano uno dei paradigmi più significativi nella progettazione di sistemi distribuiti, in quanto eliminano la necessità di un controllo centrale e sfruttano le risorse disponibili ai margini della rete (capacità di calcolo, spazio di archiviazione e banda) per fornire servizi collettivi rendendo il tutto trasparente all'utilizzatore. Nei sistemi P2P ogni nodo (o peer) svolge simultaneamente il ruolo di **client e server**, condividendo dati e servizi con altri nodi in modo cooperativo. Questa architettura consente di ottenere **scalabilità** e **tolleranza ai guasti**, ma introduce anche nuove sfide legate alla gestione della dinamicità della rete (churn), alla sicurezza e all'efficienza del routing.

A partire dai primi anni 2000, il paradigma P2P ha trovato ampia applicazione in diversi ambiti: dalle piattaforme di **file sharing** (Napster, Gnutella, eMule, BitTorrent) ai progetti di **calcolo distribuito** come SETI@home e Folding@home, fino alle **reti di streaming distribuite** (PPLive) e alle **blockchain** (Bitcoin, Ethereum). Le prime generazioni di reti P2P erano *non strutturate*, basate su collegamenti casuali tra peer e su meccanismi di ricerca come *query flooding* o *random walk*. Questi approcci garantivano una notevole semplicità dei protocolli, ma presentavano una *scarsa efficienza*, poiché il costo delle operazioni di lookup cresceva linearmente con il numero di nodi. Per superare tali limiti sono stati introdotti i **sistemi P2P strutturati**, fondati su una logica deterministica di interconnessione tra i nodi, detta **overlay network**. La forma più diffusa e studiata di overlay strutturato è la **Distributed Hash Table (DHT)** che consente di localizzare le risorse in modo efficiente e completamente decentralizzato. In una DHT, ciascun nodo gestisce una porzione dello spazio delle chiavi e mantiene informazioni soltanto su un numero limitato di vicini. Protocolli come **Chord, Pastry, Kademlia, CAN e Tapestry** hanno reso le DHT un riferimento consolidato per i sistemi distribuiti scalabili. Le tecniche introdotte, tra cui il consistent hashing e le finger tables, sono oggi impiegate in numerose piattaforme industriali come *Amazon Dynamo*, *Apache Cassandra* e *Memcached*. Le diverse implementazioni di **DHT** si distinguono principalmente in base a due fattori:

- **Hop count**: il numero medio di passaggi (*hops*) necessari affinché una richiesta raggiunga il nodo responsabile della chiave;
- **Grado del nodo**: il numero di vicini che ciascun nodo deve mantenere nella propria tabella di routing.

II. Koorde

Koorde è una Distributed Hash Table (DHT) che combina la struttura ad anello di *Chord* con le proprietà topologiche dei *grafi di De Bruijn*, al fine di ottenere un protocollo di routing efficiente e con ridotto overhead di manutenzione.

L'obiettivo del suo design è il raggiungimento dei **limiti inferiori teorici** nel compromesso tra numero di hop per lookup e grado dei nodi.

Come dimostrato da *Kaashoek* e *Karger* [**?**], gli ideatori di Koorde, in un sistema distribuito composto da n nodi e con grado massimo d, il numero minimo di passi richiesti per instradare una richiesta è nel caso peggiore $\Omega(\log_d n)$, e in media $\Theta(\log_d n - O(1))$. Ne deriva che:

- per un grado costante $d = k$, il limite inferiore ottimale per l'efficienza del routing è $O(\log n)$ hop;
- aumentando il grado a $O(\log n)$, il numero di hop può essere ridotto fino a $O(\log n / \log\log n)$.

Il legame tra grado e numero di hop è formalizzato nel *Lemma 2.1* di Kaashoek e Karger, secondo cui un sistema distribuito composto da n nodi e con grado massimo d richiede almeno $\Omega(\log_d n)$ passi di routing nel caso peggiore, e $\Theta(\log_d n - O(1))$ in media. La dimostrazione si basa su un semplice argomento combinatorio: un nodo con grado d può raggiungere al più $1 + d + d^2 + \cdots + d^h = (d^{h+1} - 1)/(d - 1)$

nodi in h passi; affinché l'intera rete di n nodi sia connessa, deve valere $d^h \geq n$, da cui segue $h \geq \log_d n$. Poiché quasi tutti i nodi risultano distribuiti a distanze prossime a questo valore, il comportamento medio del sistema è anch'esso $\Theta(\log_d n)$.

Questi risultati individuano il confine teorico tra **efficienza di routing** e **complessità di stato locale**. Le DHT a grado costante, in cui ciascun nodo mantiene lo stesso numero di collegamenti indipendentemente dalla dimensione della rete, offrono una scalabilità asintoticamente ottimale. Al contrario, esistono schemi che raggiungono un'efficienza di lookup $O(1)$, come ad esempio la *Gupta DHT*, ma al prezzo di un numero di collegamenti proporzionale al numero complessivo di nodi nella rete. Ciò implica una perdita di scalabilità per sistemi di grandi dimensioni.

Il mantenimento di un grado costante rappresenta quindi una scelta di progetto fondamentale: in reti di piccole dimensioni un grado più elevato può risultare vantaggioso in termini di latenza media, ma all'aumentare di n la **scalabilità logaritmica** dei sistemi a grado costante diventa sempre più significativa.

Dal punto di vista **architetturale**, Koorde si ispira direttamente a *Chord*. L'*overlay network* è organizzata come un **anello**, in cui lo spazio degli identificatori, per degli identificatori a b bits, è definito modulo 2^{bits}. A ciascun nodo è assegnato un **identificatore unico** all'interno di tale spazio, e lo stesso dominio di identificatori è utilizzato per associare ogni risorsa al proprio nodo responsabile. La regola di assegnazione è analoga a quella di Chord: la risorsa con identificatore k è memorizzata dal **successore** di k, ossia il primo nodo sul ring il cui identificatore è maggiore o uguale a k.

La differenza fondamentale tra *Chord* e *Koorde* risiede nella struttura e nella logica di costruzione della **tabella di routing**. In *Chord*, tale struttura è denominata **finger table**. In una rete composta da n nodi con spazio degli identificatori di dimensione 2^m, ogni nodo n_i mantiene una finger table composta da m voci, dove la j-esima voce ($1 \leq j \leq m$) punta al *successore* dell'identificatore:

$$n_i + 2^{j-1} \pmod{2^m}.$$

Questo schema consente di "saltare" esponenzialmente lungo l'anello degli identificatori, permettendo a Chord di localizzare una chiave in $O(\log n)$ passi, al costo di una tabella di routing anch'essa di dimensione $O(\log n)$ per nodo.

Koorde adotta, invece, un approccio radicalmente differente, fondato sulla topologia dei **grafi di De Bruijn**. Invece di mantenere $O(\log n)$ vicini come in Chord, ogni nodo conserva soltanto un numero costante di riferimenti (archi del grafo di De Bruijn II-A); nel caso più semplice due: il proprio *successore* sull'anello e il nodo corrispondente alla trasformazione de Bruijn del proprio identificatore. Questa costruzione permette a Koorde di simulare la connettività del grafo di De Bruijn sopra lo spazio degli identificatori di Chord, riducendo drasticamente lo stato locale mantenuto da ciascun nodo (grado costante 2) pur mantenendo un routing logaritmico con $O(\log n)$ iterazioni.

A. Grafo di de Bruijn

In teoria dei grafi, un **grafo di de Bruijn** di ordine n su un insieme di simboli $S = \{s_1, s_2, \ldots, s_m\}$ è un grafo diretto che rappresenta le relazioni di sovrapposizione tra sequenze di simboli. Tale grafo è composto da m^n vertici, corrispondenti a tutte le possibili sequenze di lunghezza n costruite sui simboli di S, ammettendo ripetizioni. Formalmente, l'insieme dei vertici è definito come:

$$V = S^n = \{(t_1, t_2, \ldots, t_n) \mid t_i \in S,\ 1 \leq i \leq n\}.$$

Un arco diretto collega due vertici $v = (t_1, t_2, \ldots, t_n)$ e $v' = (t_2, t_3, \ldots, t_n, s_j)$ se e solo se v' può essere ottenuto da v mediante lo *shift* di una posizione verso sinistra e l'aggiunta di un nuovo simbolo $s_j \in S$ in coda. L'insieme degli archi può quindi essere espresso come:

$$E = \{\big((t_1, t_2, \ldots, t_n),\ (t_2, \ldots, t_n, s_j)\big) \mid t_i, s_j \in S,\ 1 \leq i \leq n,\ 1 \leq j \leq m\}. \quad (1)$$

Da questa costruzione derivano alcune proprietà fondamentali:

- Ogni vertice possiede esattamente m archi entranti e m archi uscenti, corrispondenti alle possibili aggiunte dei simboli in S.
- Per $n = 1$, la condizione di adiacenza risulta banalmente soddisfatta per ogni coppia di vertici, producendo un grafo completo con m^2 archi.
- I grafi di de Bruijn sono sia **Euleriani** sia **Hamiltoniani**; i loro cicli euleriani e hamiltoniani coincidono con le cosiddette *sequenze di de Bruijn*.

Nel caso **binario** ($m = 2$), i vertici rappresentano tutte le possibili stringhe binarie di lunghezza n, e ogni nodo ha due archi uscenti:

$$(t_1, t_2, \ldots, t_n) \to (t_2, \ldots, t_n, 0),$$
$$(t_1, t_2, \ldots, t_n) \to (t_2, \ldots, t_n, 1).$$

Equivalentemente, se gli identificatori vengono interpretati come numeri interi, ciascun nodo m presenta due archi:

$$m \to (2m) \pmod{2^n}, \quad \text{e} \quad m \to (2m+1) \pmod{2^n}.$$

In questo modo, il grafo binario di de Bruijn rappresenta tutte le possibili transizioni fra sequenze di bit di lunghezza n, garantendo un diametro logaritmico $\log_m(m^n) = n$, e quindi un percorso massimo proporzionale alla lunghezza della sequenza.

Generalizzazione a grado $k > 2$.
Un grafo di de Bruijn può essere generalizzato a una base $k > 2$, dove ciascun vertice rappresenta una sequenza di cifre in base k e ogni nodo possiede k archi uscenti:

$$m \to (km + i) \pmod{k^n}, \quad \text{con } i \in \{0, 1, \ldots, k-1\}.$$

In questo contesto, la rappresentazione dei nodi nello spazio degli identificatori può essere vista come una codifica in base k anziché binaria. Ogni passo di routing corrisponde quindi all'aggiunta di una cifra in base k, riducendo di conseguenza la

lunghezza della sequenza necessaria per rappresentare l'intero spazio degli identificatori.

Esempio: base otto.

Considerando un grafo di de Bruijn con base $k = 8$, i vertici rappresentano tutte le possibili sequenze di tre cifre ottali, ovvero $8^3 = 512$ nodi. Ogni nodo m presenta otto archi uscenti:

$$m \to (8m + i) \pmod{8^3}, \quad i \in \{0, 1, \ldots, 7\}.$$

In tal modo, il routing può essere interpretato come una riscrittura progressiva della rappresentazione ottale dell'identificatore di destinazione. Poiché a ogni passo vengono processati tre bit (equivalenti a una cifra ottale), la lunghezza complessiva della sequenza necessaria per identificare un nodo si riduce di un fattore $\log_2 8 = 3$ rispetto al caso binario. Questa proprietà consente, a parità di spazio di indirizzamento, di diminuire il numero di hop necessari per raggiungere un nodo, migliorando l'efficienza del routing a fronte di un grado di connessione più elevato.

Tali caratteristiche rendono i grafi di de Bruijn, e le loro generalizzazioni a basi superiori, un modello estremamente flessibile per la costruzione di reti di overlay, come dimostrato dal loro impiego nel protocollo *Koorde*.

B. Routing in Koorde basato sul grafo di de Bruijn

L'utilizzo del grafo di de Bruijn in *Koorde* richiede un adattamento rispetto alla definizione teorica classica, poiché nella rete non esiste un nodo per ogni possibile identificatore dello spazio indirizzi. I nodi presenti sono infatti un sottoinsieme sparso dei 2^b identificatori possibili. Per mantenere la correttezza del modello di routing, *Koorde* associa a ciascun nodo reale anche i cosiddetti **nodi immaginari**, ossia gli identificatori di cui esso è predecessore sull'anello. In tal modo, ogni nodo si comporta come rappresentante (o *emulatore*) di tutti i nodi immaginari compresi tra il proprio identificatore (escluso) e quello del suo successore.

Durante il routing, quando un messaggio deve percorrere un arco del grafo di de Bruijn che corrisponde a un nodo immaginario non presente fisicamente nella rete, *Koorde* simula tale passaggio inoltrando la richiesta al nodo reale che è predecessore di quell'identificatore immaginario.

Un'ulteriore ottimizzazione consiste nel permettere a ciascun nodo di "impersonificare" non solo il proprio identificatore ma anche tutti i nodi immaginari di cui è predecessore. In questo modo, un messaggio può essere processato più rapidamente, poiché il nodo corrente può rappresentare più stati consecutivi del percorso de Bruijn, riducendo così il numero totale di hop necessari per completare la ricerca.

Routing in un grafo di de Bruijn binario ($k = 2$)

Nel caso binario, ogni passo di routing corrisponde all'inserimento di un singolo bit dell'identificatore di destinazione nella posizione meno significativa dell'identificatore corrente. Lo pseudocodice seguente, tratto e adattato da *Kaashoek & Karger (2003)*, mostra la procedura di lookup per un grafo di base 2:

Algorithm 1 Routing in un grafo di de Bruijn binario

```
procedure LOOKUP(m, k, kshift)
    if k = m then
        return m {Il nodo corrente è il responsabile della chiave}
    else
        t ← m ∘ topBit(kshift)
        return LOOKUP(t, k, kshift ≪ 1)
    end if
end procedure=0
```

In questa rappresentazione, l'operatore $\circ$ denota la concatenazione modulo 2^b:

$$m \circ 0 = (2m) \bmod 2^b, \qquad m \circ 1 = (2m + 1) \bmod 2^b.$$

Ad ogni iterazione viene quindi eseguito uno *shift* a sinistra dell'identificatore corrente e viene inserito un nuovo bit proveniente dalla chiave k. Dopo b iterazioni, il processo converge sul nodo responsabile della chiave.

Routing in un grafo di de Bruijn di grado k

Nel caso generale, un grafo di de Bruijn di base k consente di processare più bit per passo, inserendo una cifra in base k anziché un singolo bit. Lo pseudocodice seguente generalizza la procedura precedente:

Algorithm 2 Routing in un grafo di de Bruijn di grado k

```
procedure LOOKUP(m, k, kshift, base)
    if k = m then
        return m
    else
        d ← NextDigit(kshift, base)
        t ← (m × base + d) mod base^b
        return LOOKUP(t, k, kshift ≪ log2(base))
    end if
end procedure=0
```

In questo caso, ogni passo di routing corregge una cifra in base k dell'identificatore di destinazione, riducendo la lunghezza complessiva del percorso a $O(\log_k n)$. Un grado maggiore del grafo comporta quindi un numero minore di hop a scapito di un incremento del numero di collegamenti per nodo, offrendo un meccanismo naturale di *trade-off* tra complessità di stato locale e prestazioni di routing.

III. Implementazione

L'implementazione del protocollo *Koorde* è stata realizzata interamente in linguaggio **Go**, con particolare attenzione all'efficienza e alla modularità del codice.

La comunicazione tra i nodi della DHT è implementata mediante **gRPC**, utilizzando **Protocol Buffers (protobuf)** come formato di serializzazione dei messaggi.

A. Gli Identificatori

La gestione degli identificatori costituisce uno degli aspetti più delicati nella progettazione di una DHT. Nel caso di

Koorde, la scelta del tipo di dato per rappresentare gli identificatori è stata guidata dall'esigenza di ottenere un compromesso tra **configurabilità** e **efficienza operativa**.

A differenza di implementazioni basate su interi di lunghezza fissa (ad esempio `uint64` o `big.Int`), la presente implementazione utilizza il tipo `[]byte` (slice di byte) per rappresentare gli identificatori. Questa scelta ha richiesto la reimplementazione di operazioni aritmetiche bit a bit (*shift*, *somma modulo*, *moltiplicazione per il grado del grafo*), ma ha permesso di ottenere una gestione altamente configurabile e indipendente dal numero di bit dello spazio degli identificatori, mantenendo al contempo un'elevata efficienza.

Ogni identificatore è gestito all'interno di una struttura `Space`, che definisce i parametri dello spazio degli ID e i vincoli topologici del grafo di de Bruijn:

```
type Space struct {
    Bits         int
    ByteLen      int
    GraphGrade   int
    SuccListSize int
}
```

Listing 1. Struttura Space.

L'identificatore è quindi rappresentato dal tipo `ID`, definito come una slice di byte in formato *big-endian*, che consente di mantenere la coerenza aritmetica e lessicografica nelle operazioni di confronto e di routing:

```
type ID []byte
```

Listing 2. Definizione del tipo ID.

Le principali operazioni definite sul tipo `ID` includono:

- **Shift e moltiplicazione modulare**:

```
func (sp Space) MulKMod(a ID) (ID, error)
```

- **Somma modulo** 2^{Bits}:

```
func (sp Space) AddMod(a, b ID) (ID, error
    )
```

- **Estrazione del prossimo digit in base** k :

```
func (sp Space) NextDigitBaseK(x ID) (
    digit uint64, rest ID, err error)
```

Tali funzioni consentono di implementare in modo diretto le primitive aritmetiche necessarie per il routing descritto nelle sezioni precedenti, mantenendo la piena compatibilità con la definizione teorica di Koorde, ma con la flessibilità di una rappresentazione a lunghezza arbitraria.

L'approccio adottato garantisce quindi un'elevata configurabilità, permettendo di variare la dimensione dello spazio degli identificatori e il grado del grafo senza modificare il codice sorgente, e un'ottima efficienza nelle operazioni più frequenti, come il calcolo dei successori o dei nodi de Bruijn.

B. La Routing Table

La struttura della *routing table* in *Koorde* deriva direttamente dal modello di *Chord*, pur introducendo alcune estensioni legate alla rappresentazione del grafo di de Bruijn. In *Chord*, la dimensione della tabella è configurabile tramite un parametro iniziale, ma il grado effettivo della rete rimane costante rispetto alla dimensione del sistema, come previsto dalla teoria.

Ogni nodo mantiene un riferimento al proprio **successore**, ovvero al nodo con identificatore immediatamente successivo sul ring. Nel caso in cui sia abilitato il meccanismo di tolleranza ai guasti, la tabella include una **successor list** contenente i successori immediati fino a una lunghezza r, tipicamente $O(\log n)$. Questa struttura, introdotta da *Chord* e ereditata da *Koorde*, consente di garantire la connettività della rete anche in presenza di fallimenti: se un nodo rileva che il proprio successore non è più raggiungibile, può utilizzare il secondo successore nella lista per ristabilire la coerenza dell'anello.

In aggiunta alla successor list, *Koorde* mantiene una **lista dei nodi di de Bruijn**, che rappresenta le connessioni logiche derivate dalla topologia del grafo di routing. La dimensione di questa lista dipende dal grado k del grafo di de Bruijn utilizzato, come discusso nella Sezione II-A: ogni nodo deve mantenere informazioni sui k nodi che corrispondono agli archi in uscita dal proprio identificatore.

Nel modello teorico di *Koorde*, per un nodo m tali nodi sono definiti come i predecessori dei punti:

$$(km) \bmod 2^n, \quad (km+1) \bmod 2^n, \quad \ldots, \quad (km+k-1) \bmod 2^n.$$

In altre parole, ciascun nodo m mantiene un collegamento verso il predecessore di km e verso i successori immediati di tale nodo fino a $k-1$ posizioni. Questo approccio consente di ricostruire localmente il comportamento del grafo di de Bruijn, anche in presenza di uno spazio di identificatori sparso.

```
type RoutingTable struct {
    logger        logger.Logger
    space         domain.Space
    self          *domain.Node
    successorList []*routingEntry
    predecessor   *routingEntry
    deBruijn      []*routingEntry
}
```

Listing 3. Struttura della Routing Table in Koorde.

Ogni elemento della routing table è rappresentato da una struttura `routingEntry`, che incapsula un riferimento a un nodo e un meccanismo di sincronizzazione interna per garantire la sicurezza in presenza di accessi concorrenti. In particolare, ciascuna entry impiega un `sync.RWMutex` per fornire letture concorrenti e scritture esclusive, evitando la necessità di bloccare l'intera tabella durante le operazioni di aggiornamento o consultazione:

```
type routingEntry struct {
    node *domain.Node
    mu   sync.RWMutex
}
```

Listing 4. Gestione concorrente delle entry tramite RWMutex.

Questa strategia consente di garantire un'elevata scalabilità durante le operazioni di stabilizzazione e aggiornamento periodico della routing table, poiché più thread possono accedere

contemporaneamente a entry diverse senza bloccare l'intera struttura. La concorrenza è gestita a livello fine-grained (entry-level), riducendo la contesa tra goroutine e migliorando la reattività complessiva del nodo.

Dal punto di vista implementativo, la `successor list` viene popolata chiedendo al proprio successore la lista dei suoi successori, secondo la stessa logica adottata da *Chord*. Per la lista di de Bruijn, invece, il meccanismo teorico prevede di eseguire una `lookup` per ciascun identificatore immaginario corrispondente agli archi di uscita, e di richiedere quindi il predecessore di ogni punto calcolato $pred(km + i)$. Tuttavia, tale metodo comporta un overhead significativo, poiché ogni richiesta prevede un'operazione di lookup completa nella DHT seguita da una seconda richiesta di informazioni sul predecessore del nodo responsabile.

Poiché i nodi della rete rappresentano un sottoinsieme sparso dello spazio degli identificatori, è frequente che più identificatori immaginari condividano lo stesso predecessore reale. Una strategia più efficiente consiste quindi nel determinare il predecessore di km e nel richiedere direttamente a tale nodo la propria lista di successori, che con alta probabilità includerà anche i nodi di de Bruijn corrispondenti agli altri archi. Questo riduce sensibilmente il numero di messaggi di rete e migliora i tempi di stabilizzazione della routing table.

Un'ulteriore ottimizzazione riguarda il caso in cui il grado del grafo sia $k = 2$, ossia nel caso binario classico. In questa configurazione, è possibile evitare la memorizzazione esplicita della lista di de Bruijn e sfruttare il meccanismo di passaggio al successore durante l'algoritmo di lookup (ovvero contattando sempre $pred(2m)$). Questa scelta incrementa il numero medio di hop per operazione di lookup, ma consente di risparmiare sia uno slot nella routing table sia le chiamate periodiche di stabilizzazione necessarie per mantenere aggiornati i riferimenti.

C. Il Client Pool

Uno degli aspetti fondamentali in una *Distributed Hash Table* (DHT), e più in generale in qualunque rete *peer-to-peer*, è la gestione efficiente della comunicazione tra i nodi. In *Koorde*, tale obiettivo è perseguito attraverso una struttura dedicata, denominata **Client Pool**, che mantiene e gestisce in modo centralizzato le connessioni gRPC verso i nodi attualmente presenti nella *routing table*. Questo approccio consente di ridurre in modo significativo l'overhead di creazione e distruzione delle connessioni, favorendo il riutilizzo di canali già aperti e garantendo al contempo accesso concorrente sicuro per le varie goroutine che operano sul nodo.

Il *Client Pool* è implementato come una mappa di connessioni condivise, ciascuna associata a un indirizzo remoto e arricchita da un contatore di riferimento (*reference counter*). Il contatore rappresenta il numero di entry della *routing table* che al momento utilizzano quella connessione. Finché il contatore è maggiore di zero, la connessione rimane attiva all'interno del pool; quando il contatore scende a zero, la connessione viene chiusa e rimossa, liberando risorse di rete in modo automatico. Ogni operazione di aggiornamento della *routing table* (ad esempio l'aggiunta o la rimozione di un successore o di un nodo de Bruijn) comporta una modifica del contatore corrispondente nella struttura del *Client Pool*.

Poiché l'accesso concorrente alla mappa delle connessioni rappresenta una sezione critica estremamente ridotta rispetto alla complessità della *routing table*, si è preferito adottare un modello di sincronizzazione più semplice e leggero. In particolare, la struttura utilizza un singolo `sync.Mutex` per proteggere l'intera mappa, evitando la complessità e il costo di una gestione a grana fine.

Le strutture principali sono riportate nel Listato 5. La prima, `refConn`, incapsula una connessione gRPC con un contatore di riferimenti; la seconda, `Pool`, gestisce l'intero insieme di connessioni attive e coordina le operazioni di aggiunta, rilascio e chiusura.

```
type refConn struct {
    conn *grpc.ClientConn
    refs int
}

type Pool struct {
    selfId          domain.ID
    selfAddr        string
    lgr             logger.Logger
    mu              sync.Mutex
    clients         map[string]*refConn
    closed          bool
    failureTimeout  time.Duration
}
```

Listing 5. Strutture principali del Client Pool.

Il ciclo di vita di una connessione segue una logica di riferimento esplicita:

- all'inserimento di un nodo nella *routing table*, viene invocato il metodo `AddRef()`, che crea una nuova connessione o incrementa il contatore di una connessione esistente;
- la rimozione di un nodo dalla *routing table* comporta una chiamata a `Release()`, che decrementa il contatore; se questo raggiunge zero, la connessione viene chiusa e rimossa;
- la funzione `GetFromPool()` fornisce ai componenti della DHT un client gRPC riutilizzabile per effettuare chiamate remote in modo concorrente;
- infine, il metodo `Close()` chiude in modo sicuro tutte le connessioni ancora attive, rendendo il pool inutilizzabile.

D. Servizi gRPC integrati

La comunicazione tra i componenti della rete Koorde è interamente realizzata tramite il framework **gRPC**, con definizione delle interfacce e dei messaggi in formato **Protocol Buffers (protobuf)**. Nell'implementazione proposta sono definiti due servizi distinti:

- **Servizio interno tra nodi** (`dht/node.proto`), utilizzato per le operazioni di routing, manutenzione e replicazione della DHT;
- **Servizio esterno per i client** (`client/client.proto`), che fornisce un'interfaccia applicativa semplificata per l'accesso alle risorse e alle informazioni di rete.

1) Servizio DHT interno (`dht/node.proto`): Questo servizio rappresenta il canale di comunicazione tra i nodi della DHT e implementa tutte le primitive fondamentali di Koorde: ricerca dei successori, aggiornamento della tabella di routing e gestione delle risorse distribuite. Il servizio espone le seguenti operazioni principali:

- **FindSuccessor(FindSuccessorRequest) → FindSuccessorResponse** Implementa la procedura di *lookup* nel grafo di de Bruijn: restituisce il nodo responsabile dell'identificatore specificato in `target_id`. restituisce `Internal` se il nodo non è riuscito a proseguire il lookup.
- **GetPredecessor()** Restituisce il predecessore diretto del nodo chiamante. Se il nodo non ha ancora un predecessore, restituisce un errore `NotFound`.
- **GetSuccessorList()** Restituisce la lista dei successori immediati del nodo, utilizzata per la tolleranza ai guasti e la stabilizzazione dell'anello.
- **Notify(Node)** Notifica a un nodo che il chiamante potrebbe essere il suo predecessore. Se la notifica è valida, il nodo aggiornato modifica la propria tabella di routing di conseguenza.
- **Ping()** Operazione diagnostica che verifica la raggiungibilità di un nodo remoto, viene utilizzata per controllare la corretta attività del predecessore.
- **Store(stream StoreRequest)** Permette la memorizzazione di risorse nella DHT. Supporta streaming di richieste per l'inserimento batch di più elementi.
- **Retrieve(RetrieveRequest) → RetrieveResponse** Recupera una risorsa identificata da una chiave. Restituisce `NotFound` se la risorsa non esiste nel nodo responsabile.
- **Remove(RemoveRequest)** Rimuove una risorsa associata alla chiave specificata. Restituisce `NotFound` se la chiave è assente.
- **Leave(Node)** Gestisce la disconnessione controllata di un nodo dalla rete, notificando il successore per consentire la riallocazione delle risorse e il ripristino dell'anello.

Per le operazioni che non richiedono parametri o dati di ritorno specifici, è stato impiegato il messaggio `google.protobuf.Empty` della libreria standard di Google. Questo accorgimento consente di minimizzare il payload delle chiamate gRPC, riducendo il traffico di rete e i tempi di serializzazione/deserializzazione, specialmente nelle procedure di manutenzione periodica o ping diagnostico.

2) Servizio ClientAPI (`client/client.proto`): Il secondo servizio, **ClientAPI**, espone un'interfaccia di più alto livello rivolta ai client esterni, permettendo l'interazione con la DHT senza conoscere la topologia interna del grafo. Le operazioni principali comprendono:

- **Put(PutRequest)** Inserisce o aggiorna una risorsa nella DHT, identificata da una chiave applicativa.
- **Get(GetRequest) → GetResponse** Recupera il valore associato a una chiave. Restituisce `NotFound` se la chiave non è presente.
- **Delete(DeleteRequest)** Elimina una chiave esistente dalla DHT.
- **GetStore()** Restituisce in streaming tutte le risorse attualmente memorizzate nel nodo, per scopi diagnostici o dimostrativi.
- **GetRoutingTable()** Fornisce una rappresentazione leggibile della *routing table* del nodo, comprendente identificatore, predecessore, successori e lista di de Bruijn.
- **Lookup(LookupRequest) → LookupResponse** Esegue una lookup diretta di un identificatore arbitrario, restituendo il nodo responsabile senza coinvolgere il livello applicativo.

La separazione tra i due servizi (`DHT` e `ClientAPI`) consente di isolare la logica interna della rete dalla logica applicativa, promuovendo una chiara separazione dei ruoli:

- le interazioni **nodo–nodo** gestiscono la coerenza e la stabilità dell'overlay distribuito;
- le interazioni **client–nodo** forniscono un'API di alto livello per l'uso applicativo della DHT.

Le operazioni `GetStore()`, `GetRoutingTable()` e `Lookup()` sono state incluse unicamente a fini di **debug e analisi sperimentale del comportamento della DHT**. Tali endpoint permettono di ispezionare lo stato interno del nodo e verificare la correttezza della propagazione delle informazioni di routing durante le fasi di test o simulazione. In un contesto di produzione, tuttavia, il client dovrebbe rimanere completamente **agnostico rispetto alla struttura interna della rete** e interagire esclusivamente tramite le primitive di alto livello `Put`, `Get` e `Delete`.

E. Lo Storage locale

Nel contesto della presente implementazione, il componente di **storage** ha una funzione puramente rappresentativa. Come discusso nella sezione introduttiva, l'obiettivo del lavoro è l'analisi e la caratterizzazione del comportamento della DHT, in particolare delle dinamiche di routing e stabilizzazione, piuttosto che la persistenza dei dati. Per questo motivo, lo storage è stato implementato come un modulo *dimostrativo*, volutamente privo di meccanismi di replica, fault-tolerance o persistenza su disco.

L'implementazione fornisce un semplice archivio *in-memory* di tipo chiave–valore, protetto da un `sync.RWMutex` globale per garantire la sicurezza concorrente durante le operazioni di lettura e scrittura. Tale scelta, benché non scalabile in un contesto produttivo, è sufficiente per simulare in modo fedele le interazioni tra i nodi della DHT e verificare la correttezza del protocollo di instradamento e operazione di Leave volontaria.

```
type Storage struct {
```

```
    lgr  logger.Logger
    mu   sync.RWMutex
    data map[string]domain.Resource
}
```

Listing 6. Struttura dello storage in-memory.

L'intera struttura è mantenuta in memoria e protetta da un singolo mutex in lettura–scrittura, che garantisce accesso concorrente sicuro e consistenza dei dati senza introdurre una granularità fine di locking. In uno scenario reale, tale componente dovrebbe essere sostituito da un backend di persistenza distribuito o replicato, eventualmente integrato con meccanismi di journaling o di replica logica coerenti con le politiche di consistenza della DHT, oppure impiegato non come sistema di storage ma con altre funzionalità applicative, come quelle descritte nel paragrafo introduttivo.

F. Bootstrap e configurazione del nodo

Per garantire modularità e flessibilità nell'utilizzo del software, l'implementazione di *Koorde* prevede due diverse modalità di **bootstrap** della rete, selezionabili tramite configurazione o variabili d'ambiente. Questo approccio consente di supportare sia ambienti di test locali che scenari distribuiti su aws.

Bootstrap statico. Nella modalità statica, la configurazione del nodo può includere, sia nel file YAML sia tramite variabili d'ambiente, una lista predefinita di indirizzi `address:port`. Durante la fase di avvio, il nodo tenta di contattare un indirizzo nell'elenco per eseguire la procedura di `join`. Se la lista risulta vuota, il nodo assume automaticamente il ruolo di primo partecipante e inizializza una nuova rete *Koorde*. Questa modalità è particolarmente adatta per ambienti di test o cluster locali, dove la topologia dei peer è nota a priori.

Bootstrap tramite DNS. Per rendere il sistema *production-ready* ed eliminare ogni *single point of failure*, è stato introdotto un secondo meccanismo basato su **DNS dinamico**. In questo caso, i nodi possono utilizzare record DNS registrati su un dominio configurabile per scoprire altri partecipanti alla rete e unirsi automaticamente all'anello esistente. L'implementazione sfrutta le funzionalità messe a disposizione dal *grant AWS Route53*, consentendo la risoluzione di record SRV dinamici sia su indirizzi privati (per ambienti isolati in VPC) sia pubblici (per distribuzioni globali). L'utilizzo di indirizzi pubblici, pur offrendo uno scenario più realistico, introduce costi di risoluzione DNS superiori, poiché le query richiedono l'uscita e il rientro nella stessa rete VPC.

Meccanismo di leave volontaria. Oltre alla procedura di ingresso nella rete, è stato implementato anche un meccanismo di **uscita controllata** (*graceful leave*). Quando un nodo viene terminato, sia per spegnimento del container Docker, sia manualmente con `Ctrl+C` in esecuzione diretta, esso:

1) trasferisce tutte le risorse presenti nel proprio *storage* al successore, garantendo la continuità dei dati;
2) completa le richieste già in corso, bloccando l'accettazione di nuove richieste gRPC;
3) contatta il proprio successore per comunicare la sua leave.

In questo modo, la rete può rimanere coerente e operativa anche in presenza di rimozioni volontarie di nodi, evitando la perdita di risorse o inconsistenze nello spazio degli identificatori.

Configurabilità. Ogni nodo è completamente configurabile in tutti i suoi aspetti operativi:

- numero di bit utilizzati per lo spazio degli identificatori (`Bits`);
- grado del grafo di de Bruijn (`GraphGrade`);
- dimensione della lista dei successori (`SuccListSize`);
- modalità di bootstrap (`static` o `dns`);
- modalità di logging.

Tutti questi parametri possono essere impostati tramite file YAML per esecuzioni dirette del codice Go oppure tramite variabili d'ambiente quando il nodo è eseguito in ambiente containerizzato (Docker).

Sistema di logging. Ogni nodo dispone inoltre di un logger configurabile, progettato per fornire sia tracciabilità durante le fasi di test sia capacità diagnostiche in ambienti di produzione. Il sistema di logging supporta due modalità operative:

1) *stdout* — per la stampa diretta su terminale, utile in ambienti di sviluppo o container Docker;
2) *file mode* — per la scrittura su file persistente con gestione della rotazione;

In entrambi le modalità è possibile selezionare il livello minimo di log (`debug`, `info`, `warn`, `error`) e il formato di output (*human-readable* o *structured JSON*).

IV. Tolleranza ai guasti

Oltre alla tolleranza ai guasti dello *storage* che, come anticipato nella sezione di implementazione, non viene considerata ai fini di questo studio poiché dipende strettamente dall'ambito applicativo e dall'utilizzo specifico della DHT, l'analisi qui si concentra sui meccanismi di tolleranza ai guasti relativi alla **struttura di rete** e al **routing** della DHT basata su *Koorde*.

A. Tolleranza ai guasti della rete

Il primo meccanismo di tolleranza ai guasti riguarda la capacità del sistema di mantenere la rete connessa e coerente anche in presenza di fallimenti di nodi. Come descritto nella sezione di implementazione, *Koorde* eredita da *Chord* il meccanismo della **successor list**, che consente a ciascun nodo di mantenere una lista di successori immediati utilizzabile come *fallback* nel caso in cui il successore principale diventi irraggiungibile.

Se un nodo n_i non riceve risposta dal proprio successore diretto, può contattare il secondo successore nella lista per ristabilire la continuità dell'anello, propagando successivamente l'aggiornamento attraverso il protocollo di stabilizzazione. In questo modo, la rete rimane unita e non subisce partizionamenti, anche in presenza di guasti multipli finchè il numero di successori che si guastano contemporaneamente è $\texttt{SuccListSize} - 1$.

Formalmente, una successor list di dimensione $r = O(\log n)$ riduce la probabilità di partizionamento della rete

a valori trascurabili, poiché il fallimento simultaneo di tutti i nodi successori di un determinato nodo risulta statisticamente improbabile.

B. Tolleranza ai guasti del routing

Il secondo meccanismo riguarda la robustezza del **routing** basato sul grafo di de Bruijn. In una configurazione con grado $k = 2$, ciascun nodo mantiene un numero minimo di collegamenti di routing (due archi uscenti e due entranti). In tale scenario, se il nodo designato come prossimo hop non risponde, il sistema dovrebbe disporre di una lista di predecessori per contattare un nodo alternativo. Tuttavia, introdurre una lista di predecessori di lunghezza $O(\log n)$ per ogni nodo reintrodurrebbe la complessità e l'overhead tipici di *Chord*, annullando il vantaggio in termini di grado del nodo.

Per ovviare a questa limitazione, è possibile adottare un grafo di de Bruijn con grado maggiore, $k = O(\log n)$. In questo caso, ciascun nodo dispone di un insieme più ampio di archi uscenti, che oltre a migliorare l'efficienza media del routing ($O(\log n / \log \log n)$) forniscono una forma implicita di tolleranza ai guasti: la presenza di più possibili destinazioni per ogni passo consente di trovare percorsi alternativi anche se uno o più nodi intermedi risultano non raggiungibili.

In altre parole, la resilienza del sistema non è ottenuta tramite la replica esplicita dei collegamenti, ma emerge come proprietà strutturale del grafo di de Bruijn a grado variabile. Poiché la scelta del nodo da contattare in ciascun passo dipende dal digit estratto da `kshift`, la probabilità che il fallimento di un singolo vicino interrompa il percorso diminuisce proporzionalmente al numero di archi disponibili. In questo modo, l'aumento del grado k migliora non solo l'efficienza del routing ma anche la robustezza complessiva della rete. Il processo di *lookup* prevede, in tal caso, la selezione del nodo da contattare all'interno della lista di de Bruijn; in caso di mancata risposta o fallimento, vengono progressivamente interrogati i relativi predecessori fino all'esaurimento della lista, dopodiché il routing prosegue verso il successore come meccanismo di fallback.

V. Deploy distribuito

All'interno del repository ufficiale del progetto *KoordeDHT* sono fornite tre differenti tipologie di **deployment**, progettate per coprire scenari di sviluppo, testing e dimostrazione in ambienti distribuiti. Ogni modalità è stata strutturata per garantire una configurazione coerente e modulare, basata su file YAML e variabili d'ambiente.

A. Deploy per tracing analysis locale

In fase di progettazione era previsto un **deployment distribuito** della *Koorde DHT* su istanze *EC2*, con integrazione del servizio **AWS X-Ray** come endpoint per le chiamate *gRPC*. X-Ray avrebbe consentito di *tracciare e visualizzare il flusso delle chiamate gRPC* tra i nodi della rete, fornendo una rappresentazione dettagliata del processo di *lookup* e delle operazioni di stabilizzazione. Tuttavia, l'utilizzo di X-Ray non è stato consentito dal grant AWS disponibile, motivo per cui tale infrastruttura non è stata impiegata per analisi statistiche su larga scala, ma soltanto per verificare, in fase di sviluppo, la correttezza del flusso di lookup e del comportamento del protocollo di Koorde.

Per supportare comunque l'osservazione del sistema, viene fornito un **ambiente locale containerizzato** basato su **Docker Compose**. Questo consente di istanziare un numero arbitrario di nodi, un nodo *bootstrap* e un client interattivo, oltre al componente **Jaeger**, un sistema open-source per il *distributed tracing* che permette di raccogliere, visualizzare e analizzare le trace delle chiamate gRPC tra i nodi, misurando latenza e hop count e facilitando il debugging del routing e della stabilizzazione in ambienti controllati.

B. Deploy in cloud su singola istanza EC2

La seconda modalità di deployment è orientata al **testing distribuito** su infrastruttura cloud, utilizzando una singola istanza **AWS EC2**. In questo scenario vengono avviati multipli container Docker nella stessa macchina virtuale, simulando un cluster di nodi appartenenti alla DHT. Per introdurre condizioni di rete realistiche, il sistema integra il tool **Pumba**, che consente di applicare artificialmente *latenza*, *jitter* e *packet loss* sui container, riproducendo un contesto distribuito soggetto a imperfezioni di rete.

È inoltre presente uno **script di churn randomico**, che simula la disconnessione e la riconnessione casuale dei nodi nel tempo, permettendo di analizzare la stabilità e la resilienza della rete in condizioni dinamiche. Durante queste simulazioni viene impiegato il **client "tester"**, che genera un insieme di richieste di lookup e storage in modo casuale, con un grado di parallelismo configurabile e variabile nel tempo. Questo setup consente di misurare le prestazioni globali della rete in termini di latenza media e robustezza del routing.

C. Deploy dimostrativo multi-nodo EC2

La terza modalità di deploy ha finalità **dimostrative** e mira a riprodurre un contesto operativo realistico, in cui il **client** non possiede conoscenza diretta della topologia della rete, ma è a conoscenza del solo indirizzo di un nodo della DHT. Il deploy viene realizzato tramite l'utilizzo congiunto di **Docker Compose** e **AWS CloudFormation**, che consentono l'avvio automatizzato di **più istanze EC2**, ciascuna contenente **multipli container** della DHT cooperanti all'interno di una **VPC** logica comune.

In questo scenario, il client eseguito in locale può connettersi tramite l'**indirizzo pubblico** di una delle istanze EC2 e inviare richieste *put*, *get*, *delete* e *lookup* verso la rete distribuita, simulando un'interazione reale da parte di un'applicazione esterna.

Per garantire robustezza e disponibilità, in questa configurazione viene inoltre impiegato il servizio **AWS Route 53**, che funge da sistema di **DNS dinamico**. Tutti i nodi registrano automaticamente il proprio indirizzo come *bootstrap node*, eliminando così il rischio di un *single point of failure* e assicurando una fase di bootstrap completamente decentralizzata.

VI. Valutazione sperimentale

A. Configurazione dei test

Sono stati eseguiti numerosi test sperimentali variando progressivamente diversi parametri strutturali e operativi della rete, al fine di valutare la stabilità e la resilienza del protocollo **Koorde** in differenti condizioni operative. In particolare, sono stati modificati:

- la **dimensione dell'identificativo** (numero di bit);
- il **grado del grafo di de Bruijn**;
- la **lunghezza della successor list**;
- la **durata complessiva della simulazione**;
- il **tasso di richieste di lookup** e il relativo **grado di parallelismo**;
- i **parametri di rete** (delay medio e jitter).

Tra le varie configurazioni testate, si riporta di seguito una delle più stressanti, selezionata come caso rappresentativo per analizzare il comportamento del sistema in condizioni di forte instabilità.

B. Configurazione stress-test

La configurazione scelta prevede una rete composta da **50 nodi**, con le seguenti caratteristiche:

- **grado del grafo di de Bruijn pari a 8**;
- **successor list abilitata**;
- **identificativi a 66 bit**;
- **tasso di churn elevato**, con un evento ogni **30 secondi** e probabilità equivalente di join e leave, senza periodi di stabilità tra un evento e l'altro;
- **latenza di rete elevata**, con **100 ms di delay medio**, **50 ms di jitter** e una **probabilità non nulla di perdita di pacchetti**.

C. Risultati e osservazioni

TABLE I
RISULTATI DELLE RICHIESTE DI LOOKUP NELLA CONFIGURAZIONE STRESS-TEST

Esito	Numero di richieste	Percentuale (%)
SUCCESS	1894	86.9
TIMEOUT	248	11.4
ERROR (gRPC interno)	16	0.7
Totale richieste: 2158		

[a]I valori percentuali sono calcolati sul totale delle richieste analizzate.

I risultati ottenuti sono in linea con le aspettative teoriche. Si osserva che la maggior parte delle richieste di lookup presenta tempi di risposta contenuti, segno che il meccanismo di routing riesce a individuare rapidamente un nodo vicino alla chiave di destinazione. Le latenze residue, dell'ordine di circa **2000 ms**, risultano coerenti con il tempo medio di propagazione atteso per la configurazione adottata, in particolare per identificativi a 66 bit e grado 8 del grafo di de Bruijn, che comportano in media circa **22 salti di routing**.

Alcune richieste risultano **abortite** per superamento del timeout di **10 secondi**, un comportamento previsto considerando l'elevato churn, la presenza di ritardi di rete e il tasso di richieste di circa **0.8 lookup al secondo** con un **grado di parallelismo compreso tra 1 e 5 richieste simultanee**.

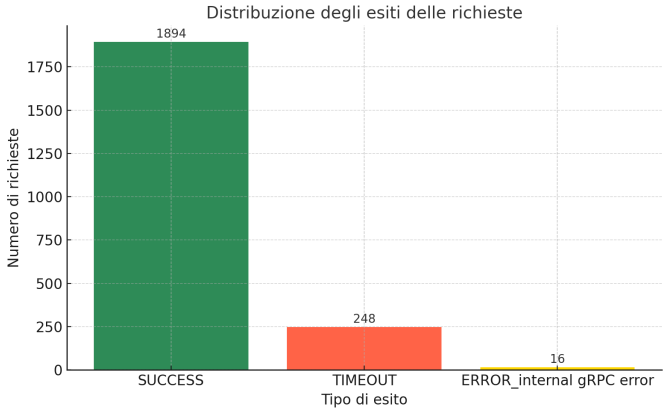

Fig. 1. Distribuzione degli esiti delle richieste.

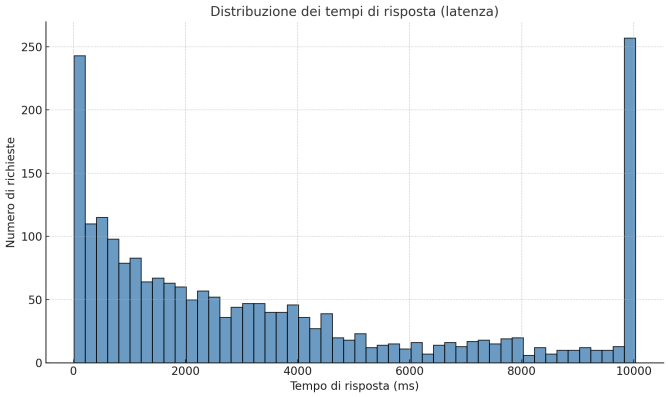

Fig. 2. Distribuzione dei tempi di risposta.

D. Conclusioni sperimentali

Nel complesso, l'esperimento conferma che l'implementazione di **Koorde** mantiene **stabilità e correttezza del routing** anche in condizioni estreme di variabilità topologica e di rete. L'aumento del churn e della latenza influisce principalmente sulla variabilità temporale dei tempi di risposta, ma non compromette la capacità del sistema di completare le lookup correttamente, evidenziando la robustezza dei meccanismi di **successor list** e del **routing basato su grafo di de Bruijn**.