# Binary Tree DHT

Alessandro Piccione
*Corso di Sistemi Distribuiti e Cloud Coputing*
*Università degli studi di Roma "Tor Vergata"*
Matricola: 0365631
alessandro.piccione.02@alumni.uniroma2.eu

***Abstract*—Questo lavoro descrive la progettazione, l'implementazione e la valutazione di una Distributed Hash Table (DHT) basata su una struttura ad albero binario per la gestione dinamica dei nodi e delle risorse. La DHT supporta le principali operazioni di inserimento (PUT) e prelievo (GET) delle risorse, nonché la gestione dei churn attraverso meccanismi di join e leave dei nodi. L'approccio proposto garantisce una distribuzione efficiente delle chiavi, la consistenza della rete e la resilienza rispetto a ingressi e uscite dinamiche dei nodi.**

## I. Introduzione

Le Distributed Hash Table (DHT) rappresentano un paradigma fondamentale degli ultimi decenni per la gestione distribuita di risorse in reti P2P. Quest'ultime consentono di definire una strategia per mappare la responsabilità delle chiavi ai nodi della rete, garantendo un accesso efficiente e scalabile. Esistono diverse implementazioni di Distributed Hash Table (DHT). Tra le più note vi sono Chord, che utilizza il consistent hashing per gestire le responsabilità dei nodi e le finger table per le operazioni di lookup, e Kademlia, che sfrutta la metrica XOR e i k-buckets per un routing distribuito e simmetrico. Altre implementazioni, meno note, includono BATON, che, analogamente all'algoritmo proposto in questo lavoro, impiega alberi binari per gestire il churn e le operazioni di inserimento e prelievo. L'algoritmo presentato in questo studio combina alcuni elementi di queste precedenti soluzioni, come l'algebra dei moduli per calcolare gli ID dei nodi e delle chiavi delle risorse, e l'utilizzo degli alberi binari per garantire la gestione dinamica delle responsabilità dei nodi e delle risorse.

## II. DHT Binary Tree

### A. L'albero binario

Nel seguente sistema abbiamo un inisieme di peer che cooperano per mantenere in modo scalabile e resiliente ai churn una Hashing Table distribuita. Ciascun peer è a sua volta un nodo di un albero binario. Dove l'albero binario è un grafo aciclico in cui ogni nodo può avere al massimo due nodi figli, comunemente detti figlio sinistro e figlio destro. Il nodo dell'albero binario può essere di tre tipologie:

- Nodo radice: è il primo nodo accessibile dell'albero. Questo nodo non possiede un nodo padre, ma può possedere due figli.
- Nodo interno: è un nodo che ha sia un padre che dei figli.
- Nodo foglia: è un nodo che ha solo il padre ma non dei figli.

Nell'ambito della seguente DHT utilizzeremo questa struttura gerarchica per effettuare operazioni di lookup e generare delle routing table. Potrebbe sorgere spontanero chiedersi, perché utilizzare una struttura ad albero binario. La risposta risiede nel fatto che l'altezza di un albero binario perfettamente bilanciato costituito da n nodi è pari a:

$$h = \log_2(n)$$

Quindi se abbiamo un albero binario perfettamente bilanciato, possiamo esplorarlo in altezza con un costo logaritmico.

### B. Operazioni fondamentali

L'algoritmo si propone di implementare quattro operazioni fondamentali.

- Join: l'operazione di join consente a un nuovo nodo di entrare nella rete e di partecipare alla gestione della DHT. Nella progettazione di questa funzionalità, è fondamentale garantire che le risorse assegnate ai nodi prima dell'ingresso del nuovo nodo non vengano ridistribuite.
- Leave:l'operazione di leave consente a un nodo di uscire dalla rete mantenendo la consistenza della Distributed Hash Table (DHT). Durante questa operazione, le chiavi gestite dal nodo uscente devono essere trasferite a un nodo appropriato, minimizzando l'impatto sulla ridistribuzione complessiva delle risorse.
- Put: l'operazione put permette ad un nodo di inserire all'interno della DHT una risorsa costituita da una chiave ed un valore. La chiave deve essere tale da identificare univocamente la risorsa. Questa operazione deve andare alla ricerca del nodo che assumerà la responsabilità della risorsa.
- Get: L'operazione get permette, a partire dalla sola chiave di una risorsa, di determinare il nodo responsabile tramite la funzione di hash e/o il meccanismo di routing della DHT, inoltrare la richiesta fino a raggiungere il nodo detentore e prelevare la risorsa, restituendola al nodo richiedente.

### C. Implementazione

In questa sezione verrà descritta l'implementazione delle operazioni elencate al passo precedente. Andremo per ciascuna di queste a vedere il funzionamento in linea generale, per poi passare allo pseudocodice. Essendo ogni nodo della DHT un servent, ovvero un componente che agisce sia da client che da server, ci saranno due blocchi di pseudocodice; il primo servirà

a descrivere il comportamento del client e quali tipologie di richieste invia. Mentre il secondo blocco descriverà il funzionamento della parte server che riceverà le richieste.
Quando un nodo vuole entrare all'interno della rete calcola:

$$y = H(\text{meta}) \bmod 2^m$$

Dove meta sono le metainformazioni relative al peer che vuole fare la join. A partire da $y$, viene costruito iterativamente un percorso verso il nodo root tramite l'operazione:

$$y_i = \frac{y_{i-1} - 1}{2}$$

finché non si raggiunge 0, che corrisponde al nodo root. Ad esempio, se $y = 12$, il percorso sarà:

$$0 \to 2 \to 5 \to 12$$

Durante il join, il nuovo nodo contatta il nodo root (assunto sempre disponibile) e poi i nodi intermedi lungo il percorso. Se un nodo del percorso non esiste, oppure un nodo durante il percorso necessita di un figlio, il nuovo nodo riceve l'identificatore relativo alla posizione mancante; in caso contrario, continua lungo il percorso fino a trovare una posizione disponibile. Se il nodo finale è già presente, viene segnalato un errore di ID già preso.
Di seguito viene mostrata l'implementazione lato client e lato server dell'algoritmo di join con lo pseudocodice.

**Algorithm 1:** Join dal lato Client

```
Function ClientJoin(meta_info_nodo)
    x ← H(meta_info_nodo);
    y ← x mod 2^m;
    percorso ← [];
    while y > 0 do
        aggiungi y a percorso;
        y ← ⌊(y − 1)/2⌋;
    aggiungi 0 a percorso;
    currAddr ← indirizzo del nodo con id 0
    foreach nodo in percorso dall'inizio do
        invia richiesta JOIN a nodo;
        if RESP=="NEED CHILD" then
            assegna ID al nuovo nodo;
            aggiorna tabella di routing;
            interrompi loop;
        currAddr ← indirizzo del nodo successivo
    if nessun nodo disponibile then
        segnala errore: ID già preso;
```

**Algorithm 2:** Gestione Join lato Server

```
Function ServerJoin(richiesta_join)
    ricevi richiesta JOIN dal client;
    if posizione figlio libera then
        assegna ID al nodo richiedente;
        aggiorna tabella di routing locale;
        restituisci "NEED CHILD";
    else
        determina nodo successivo lungo il percorso;
        inoltra richiesta JOIN al nodo successivo;
```

Quando un nodo lascia la rete, trasferisce la propria tabella di routing — contenente i riferimenti ai nodi figli — al nodo padre, che ne assume temporaneamente le responsabilità di gestione e di custodia delle risorse. Nel caso in cui il nodo uscente abbia dei figli, questi devono aggiornare il proprio riferimento al nuovo nodo padre, puntando al "nodo nonno". La situazione permane fino all'ingresso di un nuovo nodo che occupa la posizione lasciata libera, momento in cui il nodo padre restituisce le risorse e i riferimenti ereditati.

**Algorithm 4:** Leave - Lato Server

```
Data: Nodo padre P, riceve LEAVE_NOTICE(id_N, RT_N, R_N)
Result: Gestione temporanea delle risorse e
        aggiornamento della tabella di routing
begin
    Registra che il nodo id_N è uscito dalla rete;
    Memorizza temporaneamente le risorse R_N;
    Integra i riferimenti della tabella di routing RT_N
     nella propria RT_P;
    if il nodo id_N aveva figli then
        Aggiorna la tabella di routing: imposta P come
         nuovo padre dei figli di id_N;
    Mantieni le risorse e i riferimenti in custodia fino
     all'arrivo di un nuovo nodo;
```

**Algorithm 3:** Leave - Lato Client

```
Data: Nodo N con identificativo id_N, tabella di
      routing RT_N, insieme di risorse R_N
Result: Notifica di uscita e trasferimento delle
        responsabilità al nodo padre
begin
    if N è il nodo root then
        Stampa messaggio di errore: "Il nodo
         root non può lasciare la rete";
        return;
    Identifica il nodo padre P;
    Invia a P un messaggio LEAVE_NOTICE(id_N,
     RT_N, R_N);
    Libera tutte le risorse locali;
    Arresta il processo del nodo;
```

Se un nodo volesse inserire una risorsa all'interno della DHT, allora viene calcolato:

$$KEY = H(\text{meta\_risorsa}) \bmod 2^m$$

Dove meta_risorsa indica le metainformazioni della risorsa di cui vogliamo fare l'inserimento.
A partire da KEY, viene costruito iterativamente un percorso verso il nodo root tramite l'operazione:

$$y_i = \frac{y_{i-1} - 1}{2}$$

Quindi se ad esempio la chiave in rappresentazione decimale si trova sotto la forma y=12, il percorso sarà:

$$0 \to 2 \to 5 \to 12$$

Una volta ricostruito il percorso, il nodo che vuole effettuare l'inserimento contatta prima il nodo root e poi a seguire tutti gli altri nodi che si ritrova all'interno del percorso. Ogni volta i nodi contattati vanno a vedere se esiste il prossimo nodo nel percorso, se quest'ultimo non dovesse esistere allora memorizza la risorsa nella propria hash table, salvando la chiave KEY ed il valore associato.

**Algorithm 5:** PUT - Lato Client

**Data:** Risorsa $R$ con metadati `meta_risorsa`
**Result:** Inserimento della risorsa nella DHT
**begin**
- Calcola la chiave: $KEY \leftarrow H(\text{meta\_risorsa}) \bmod 2^m$;
- Costruisci il percorso $path$ applicando iterativamente: $y_i = \lfloor (y_{i-1} - 1)/2 \rfloor$;
- Imposta $currentNode \leftarrow root$;
- **for** *ogni* $nextNode$ *in* $path$ **do**
  - Invia richiesta `PUT_REQUEST(`$KEY$`, `$R$`, `$nextNode$`)` a $currentNode$;
  - **if** *riceve risposta* `STORE_CONFIRM` **then**
    - Stampa "`Risorsa memorizzata in`" $currentNode$;
    - **return**;
  - **else if** *riceve risposta* `FORWARD(`$nextNode$`)` **then**
    - Aggiorna $currentNode \leftarrow nextNode$;

**Algorithm 6:** PUT - Lato Server

**Data:** Nodo $N$ che riceve `PUT_REQUEST(`$KEY$`, `$R$`, `$target$`)`
**Result:** Memorizzazione o inoltro della risorsa
**begin**
- **if** *il nodo* $target$ *esiste nella tabella di routing* **then**
  - Inoltra la richiesta a $target$;
  - Rispondi con `FORWARD(`$target$`)`;
- **else**
  - Memorizza localmente la risorsa $R$ nella hash table di $N$;
  - Registra la chiave $KEY$ e il valore associato;
  - Rispondi al mittente con `STORE_CONFIRM`;

L'operazione di prelievo prende in input la chiave KEY che identifica la risorsa e sulla base di ciò come abbiamo visto precedentemente va a calcolarsi un percorso. A questo punto il nodo che vuole prelevare una risorsa effettua una chiamata a tutti i nodi con gli id presenti nel percorso ricostruito. Il nodo che riceve questa chiamata controlla se è in possesso della risorsa. Se dovesse esserlo allora restituisce la risorsa altrimenti restituisce l'indirizzo per contattare il nodo figlio presente nel percorso.

**Algorithm 7:** GET - Lato Client

**Data:** Chiave $KEY$ della risorsa da prelevare
**Result:** Restituisce la risorsa corrispondente a $KEY$
**begin**
- Costruisci il percorso $path$ applicando iterativamente: $y_i = \lfloor (y_{i-1} - 1)/2 \rfloor$;
- Imposta $currentNode \leftarrow root$;
- **for** *ogni* $nextNode$ *in* $path$ **do**
  - Invia richiesta `GET_REQUEST(`$KEY$`)` a $currentNode$;
  - **if** *riceve risposta* `RESOURCE(`$R$`)` **then**
    - Restituisci $R$;
    - **return**;
  - **else if** *riceve risposta* `FORWARD(`$nextNode$`)` **then**
    - Aggiorna $currentNode \leftarrow nextNode$;
- Stampa "`Risorsa non trovata nella rete`";

**Algorithm 8:** GET - Lato Server

**Data:** Nodo $N$ che riceve `GET_REQUEST(`$KEY$`)`
**Result:** Restituisce la risorsa se presente oppure inoltra al figlio
**begin**
  **if** *il nodo $N$ possiede la risorsa $KEY$* **then**
    Restituisci `RESOURCE(`$R$`)` al mittente;
  **else**
    Determina $nextNode$ seguendo il percorso della chiave $KEY$;
    Restituisci `FORWARD(`$nextNode$`)` al mittente;

## III. Testing e valutazione

Per verificare il corretto funzionamento della Distributed Hash Table (DHT) implementata, è stato eseguito un test basato sul conteggio degli hop necessari per completare le operazioni di `PUT` e `GET`. In questo contesto, un *hop* corrisponde al passaggio di una richiesta da un nodo all'altro lungo il percorso calcolato tramite l'algoritmo basato sugli alberi binari.

L'obiettivo principale del test è stato valutare l'efficienza della rete nella gestione delle richieste, in particolare in termini di numero di nodi attraversati per raggiungere il nodo responsabile della chiave o della risorsa. Questo parametro fornisce un'indicazione diretta della complessità logaritmica della DHT e della bontà della strategia di instradamento implementata.

I test sono stati eseguiti simulando l'inserimento di nuove risorse (`PUT`) e il loro recupero (`GET`) in diversi scenari, variando il numero di nodi presenti nella rete. Per ciascuna operazione è stato registrato il numero di hop necessari, permettendo di osservare come l'algoritmo gestisca il percorso iterativo verso i nodi target.

Dall'analisi emerge che il numero di hop è logaritmico rispetto al numero di nodi quanto più l'albero è bilanciato. Questo comportamento è coerente con quanto atteso dalla teoria degli alberi binari, in cui la profondità dell'albero determina il numero massimo di passaggi necessari per raggiungere qualsiasi nodo. Inoltre, il test ha permesso di osservare come la rete gestisca dinamicamente l'inserimento e la rimozione dei nodi, confermando la correttezza delle operazioni di JOIN e LEAVE in termini di mantenimento delle responsabilità sulle chiavi e sulla tabella di routing.

## IV. Osservazioni e possibili miglioramenti

Dall'analisi della Distributed Hash Table implementata emergono alcune considerazioni interessanti riguardo al comportamento dell'algoritmo e alle sue prestazioni. In primo luogo, l'utilizzo della struttura ad albero binario per la gestione delle operazioni di `JOIN`, `LEAVE`, `PUT` e `GET` ha dimostrato di garantire un numero di hop relativamente basso, confermando la complessità logaritmica attesa e riducendo al minimo la latenza complessiva nella rete.

Un altro aspetto positivo osservato è la capacità dell'algoritmo di gestire il churn in maniera efficiente. L'impatto della redistribuzione delle chiavi durante l'ingresso e l'uscita dei nodi si è rivelato limitato, permettendo di mantenere una consistenza stabile della DHT anche in presenza di nodi dinamici. Questo comportamento rappresenta un punto di forza significativo rispetto ad altre implementazioni più tradizionali che richiedono una ridistribuzione massiva delle risorse.

Tuttavia, alcune criticità possono essere considerate per futuri miglioramenti. Ad esempio, in scenari con un numero molto elevato di nodi, l'algoritmo potrebbe beneficiare dell'introduzione di meccanismi di caching o di routing aggiuntivi per ridurre ulteriormente il numero di hop medio, specialmente durante operazioni di `GET` su chiavi distribuite nei nodi più profondi dell'albero. Inoltre, la gestione dei nodi root e dei nodi intermedi potrebbe essere ottimizzata per aumentare la resilienza complessiva della rete in caso di fallimenti simultanei.

Infine, l'integrazione di metriche di bilanciamento del carico tra i nodi potrebbe contribuire a distribuire in maniera più equa le responsabilità delle chiavi, migliorando ulteriormente le performance della DHT in contesti altamente dinamici. Questi possibili miglioramenti rappresentano spunti concreti per futuri sviluppi e ottimizzazioni dell'algoritmo, mantenendo al contempo la semplicità e la scalabilità della struttura ad albero binario.