



1506  
UNIVERSITÀ  
DEGLI STUDI  
DI URBINO  
CARLO BO

# Implementazione di Kademlia in Erlang

Corso: Applicazioni Distribuite e Cloud Computing

May 28, 2025

**Docente:** Prof. Claudio Antares Mezzina

**Studente:** Andrea De Lorenzis

# Contents

<b>1</b>	<b>Introduzione</b>	<b>3</b>
<b>2</b>	<b>Kademlia</b>	<b>3</b>
2.1	Routing . . . . .	4
2.2	Protocollo . . . . .	5
2.3	Ripubblicazione e scadenza dei valori . . . . .	6
2.4	Caching dei valori . . . . .	6
2.5	Refresh dei bucket . . . . .	6
2.6	Join nella rete . . . . .	7
<b>3</b>	<b>Scelte di progetto</b>	<b>7</b>
3.1	Stato di un nodo . . . . .	7
3.2	Bootstrap di un nodo . . . . .	9
3.3	Republish e Refresh . . . . .	9
<b>4</b>	<b>Implementazione</b>	<b>9</b>
4.1	Struttura del progetto . . . . .	9
4.2	Avvio della rete . . . . .	10
4.3	Creazione e inizializzazione dei nodi . . . . .	11
4.4	Funzionamento di un nodo . . . . .	14
4.5	Logica di lookup . . . . .	16
4.6	Republish dei valori . . . . .	21
4.7	Refresh dei bucket . . . . .	23
4.8	Funzioni ausiliarie . . . . .	24
<b>5</b>	<b>Testing</b>	<b>27</b>
5.1	Creazione rete e nodi . . . . .	27
5.2	Operazioni di protocollo . . . . .	29
5.3	Republish dei valori . . . . .	31
5.4	Refresh dei bucket . . . . .	32
<b>6</b>	<b>Risultati sperimentali</b>	<b>32</b>
6.1	Tempo di lookup medio . . . . .	32
6.2	Tempo di join medio . . . . .	33
6.3	Lookup in presenza di nodi guasti . . . . .	34
<b>7</b>	<b>Conclusioni</b>	<b>36</b>

# 1 Introduzione

## 2.1 Routing

Il routing in Kademlia è iterativo, ossia il nodo iniziatore gestisce l'invio di tutte le query, mano a mano che riceve nodi più vicini al target. Ciò è diverso dal routing ricorsivo in cui sono i nodi intermedi che si occupano di mandare avanti la ricerca.

La distanza tra due ID nel overlay è definita tramite la metrica XOR. Dati due ID  $x$  e  $y$ , la distanza in senso XOR è definita come l'intero:

$$d(x, y) = x \oplus y.$$

La metrica si sposa bene con la rappresentazione ad albero binario del sistema. La foglia più vicina ad un ID  $x$  è la foglia/nodo il cui ID condivide con  $x$  il prefisso più lungo.

Il routing in Kademlia garantisce il raggiungimento del nodo target in circa  $\log_2 N$  passi (hop), dove  $N$  è il numero di nodi nella rete. L'immagine in figura 2 mostra una raffigurazione del processo di lookup e di come il suo andamento sia logaritmico.

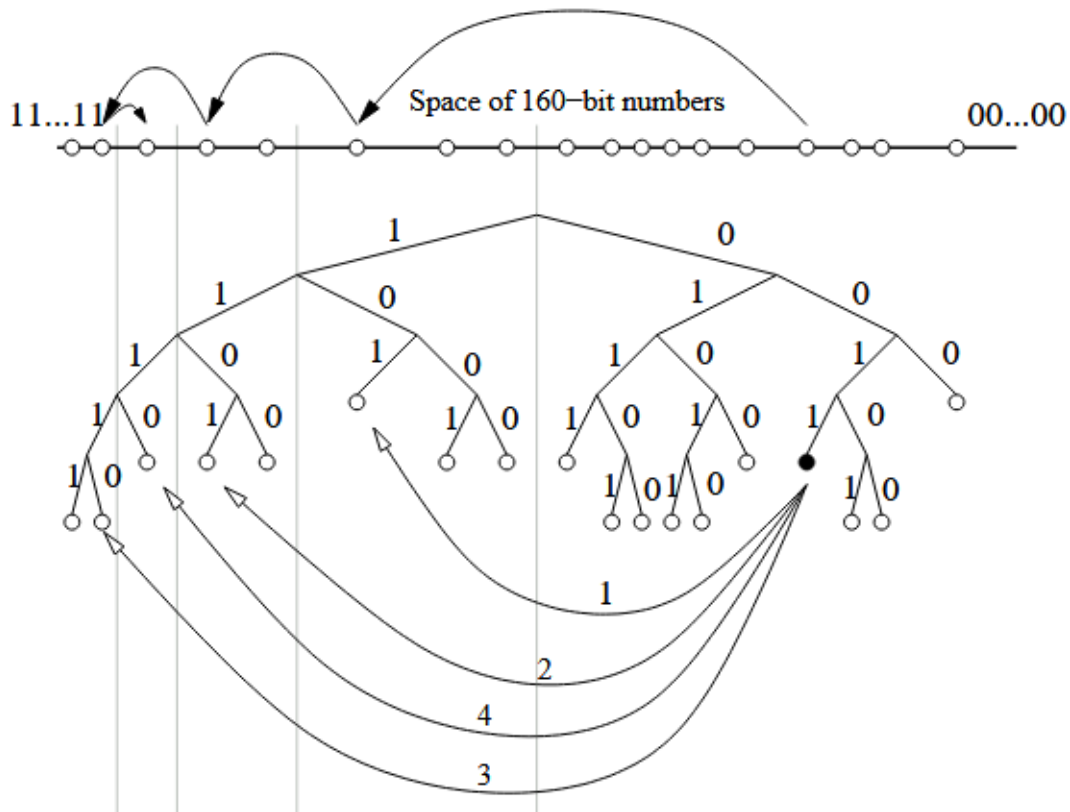


Figure 2: Lookup di un nodo tramite il suo ID. Il routing avviene contattando nodi sempre più vicini al ID target. La linea in cima rappresenta lo spazio degli ID a 160 bit. Ad ogni passo si percorre una porzione significativa dello spazio, convergendo così velocemente al target.

Ciascun nodo mantiene nel proprio stato una tabella di routing con le informazioni sugli altri contatti. Come detto, ogni nodo deve avere almeno un contatto in uno dei sottoalberi che non lo contiene. Questo viene fatto tramite i cosiddetti *k-bucket*. Un *k-bucket* è una lista di nodi, dove ogni elemento è una tripla `<IP address, UDP port, Node ID>`. Nella tabella di routing ci sono  $M$  *k-buckets*, dove  $M$  è il numero di bit nel keyspace (e.g. 160). Ciascun bucket rappresenta un sottoalbero. I primi bucket corrispondono ai sottoalberi

più grandi non contenenti il nodo, mentre gli ultimi bucket sono i sottoalberi più piccoli e vicini al nodo di riferimento. Questo partizionamento viene descritto con un esempio in figura 3.

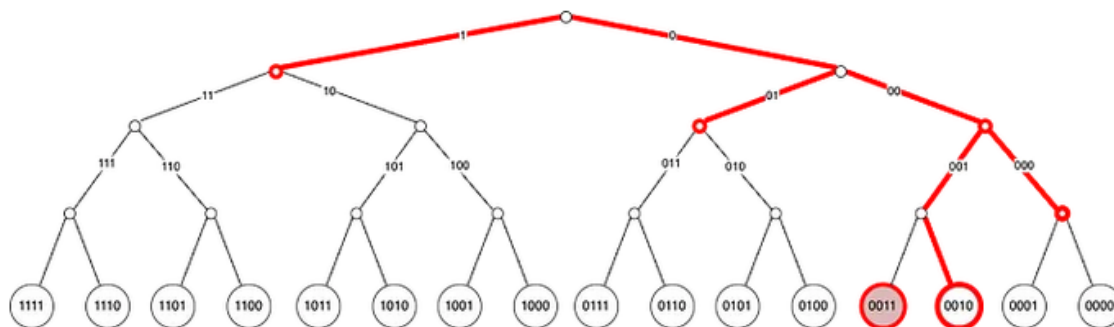


Figure 3: Albero binario degli ID Kademlia senza rami mancanti. Gli ID hanno lunghezza 4, quindi ci sono quattro bucket. Nell'albero sono evidenziati in rosso i prefissi che corrispondono ai bucket del nodo 0011 sotto osservazione. Il primo bucket corrisponde a zero bit in comune (1xxx), il secondo bucket corrisponde a un bit in comune (0xxx), il terzo a due (00xx), il quarto a tre bit in comune (001x), infine c'è il nodo stesso 0011.

Un  $k$ -bucket contiene fino a  $k$  contatti, dove  $k$  è un parametro globale di replicazione. I  $k$ -bucket "più vicini" conterranno in generale meno contatti, perché esistono meno nodi vicini al nodo in questione, mentre quelli "lontani" saranno maggiormente popolati.  $k$  viene scelto in maniera tale che la probabilità che tutti i  $k$  nodi falliscano nel giro di un'ora sia molto bassa, solitamente si usa  $k = 20$ .

I  $k$ -bucket sono ordinati in base al momento di ultimo contatto; i nodi contattati più di recente vengono messi in fondo alla lista, quello contattato meno recentemente invece è mantenuto in testa alla lista. Quando un nodo riceve un messaggio da un altro, aggiorna il corretto  $k$ -bucket, selezionato sulla base dell'ID del mittente. Se il bucket ha meno di  $k$  entrate, aggiunge il contatto in fondo; se il bucket è pieno, invia un ping al nodo in testa (quello contattato meno di recente). Se il nodo contattato risponde alla ping, allora questo viene mantenuto e spostato in fondo alla lista; altrimenti, si scarta quest'ultimo e si aggiunge il nuovo nodo in fondo alla lista. Si ha quindi una politica di *least-recently eviction*, con mantenimento però dei nodi fintanto che rimangono attivi.

## 2.2 Protocollo

Il protocollo Kademlia è formato da quattro chiamate di procedure remote (RPC):

- **PING**: sonda un altro nodo per capire se è ancora attivo;
- **STORE**: comunica al nodo di salvare una coppia <chiave, valore>;
- **FIND\_NODE**: comunica al nodo di restituire i  $k$  nodi più vicini ad un target ID. Questi possono essere presi da un singolo bucket o da più bucket se il primo ha meno di  $k$  contatti. Se in tutto ci sono meno di  $k$  bucket nella tabella di routing, il nodo restituisce tutti i nodi di cui è a conoscenza;
- **FIND\_VALUE**: funziona in maniera simile a **FIND\_NODE**, ma se il nodo ha salvato precedentemente il valore, allora restituisce solo quello.

L'operazione più importante è quella che consiste nel localizzare i  $k$  nodi più vicini ad un certo ID (*node lookup*). Il nodo iniziatore sceglie  $\alpha$  nodi più vicini dalla sua tabella di routing, dove  $\alpha$  è un parametro globale di parallelismo (e.g.  $\alpha = 3$ ). Questi possono provenire da un singolo bucket se ha almeno  $\alpha$  contatti, o da più bucket. L'iniziatore invia poi delle *FIND\_NODE* parallele e asincrone agli  $\alpha$  nodi selezionati. Quando riceve risposta da una di queste RPC, seleziona dai nodi raccolti gli  $\alpha$  nodi più vicini che non ha ancora contattato, e gli invia delle nuove *FIND\_NODE*. Se un round di  $\alpha$  *FIND\_NODE* non porta a nessun nodo più vicino rispetto al più vicino dei nodi già visitati, allora parte un ultimo round di *FIND\_NODE* verso i  $k$  nodi più vicini non ancora contattati.

Quasi tutte le operazioni sono implementate sfruttando questa procedura di node lookup. Ad esempio, quando si vuole salvare una coppia chiave-valore, si localizzano innanzitutto i  $k$  nodi più vicini, e poi gli si invia la RPC *STORE*.

### 2.3 Ripubblicazione e scadenza dei valori

Ogni nodo ripubblica le coppie periodicamente, garantendo la persistenza dei valori nella rete nel corso del tempo. Questo viene fatto perché, se i valori non venissero ripubblicati periodicamente, potrebbero perdersi quando alcuni dei  $k$  nodi che possedevano il valore si disattivano, oppure quando entrano nuovi nodi che sono più vicini alla chiave dei possessori attuali. In entrambi i casi, senza ripubblicazione, i lookup per i valori inizierebbero a fallire dopo un po' di tempo.

Per evitare di congestionare la rete, quando uno dei nodi che possiede il valore riceve una *STORE*, evita di ripubblicare quel valore nell'ora successiva. In questo modo, fintanto che i nodi non hanno orologi perfettamente sincronizzati, solo un nodo alla volta dei  $k$  ripubblica una certa coppia ogni ora.

Per ciascun valore è assegnata una scadenza, in modo da eliminare i dati troppo vecchi, e quindi potenzialmente inutili, dalla rete. Periodicamente, il publisher originale di un valore lo andrà a ripubblicare nella rete, aggiornando così il tempo di scadenza sui  $k$  nodi più vicini a quel valore.

### 2.4 Caching dei valori

Per trovare una coppia chiave-valore si parte facendo un lookup dei  $k$  nodi più vicini alla chiave. Invece di mandare delle *FIND\_NODE*, si inviano delle *FIND\_VALUE* durante il processo di lookup. In questo modo, la procedura si arresta non appena uno dei nodi contattati restituisce il valore desiderato. Per migliorare l'efficienza, si può inserire un meccanismo di caching secondo cui, non appena si trova un valore tramite value lookup, lo si salva sul nodo più vicino alla chiave che non ha restituito il valore tra quelli visitati. Data l'unidirezionalità della topologia, future ricerche per quella chiave da parte di tutti i nodi avranno un'alta probabilità di raggiungere queste entrate di cache prima di quelle originali. Per evitare di saturare la rete con valori di cache, si rende il tempo di scadenza di una coppia esponenzialmente inversamente proporzionale al numero di nodi tra il nodo corrente e il nodo più vicino alla chiave. Questo numero lo si può calcolare dalla tabella di routing del nodo corrente.

### 2.5 Refresh dei bucket

I bucket sono aggiornati dal continuo traffico di richieste che viaggiano attraverso i nodi. Per gestire casi in cui non ci sono richieste per un certo intervallo di ID (e quindi i bucket corrispondenti sono meno popolati) ogni nodo aggiorna tutti i bucket per cui non ha svolto

una node lookup nell'ora precedente. Per aggiornare un bucket, si genera un ID casuale che ricada in quel bucket e si esegue una `FIND_NODE` per quel ID, salvando tutti i nuovi risultati ottenuti all'interno della tabella di routing del nodo.

## 2.6 Join nella rete

Per entrare a far parte del overlay, un nodo ha bisogno di contattare un altro nodo già partecipante, che prende il nome di nodo di *bootstrap*. Il nodo entrante inserisce il nodo di bootstrap nella propria tabella di routing. Poi, esegue una node lookup per il proprio ID. In questo modo popola la propria tabella di routing con i nodi a lui più vicini, e si fa conoscere a sua volta da questi nodi. Infine, esegue un refresh di tutti i bucket, per popolare anche i bucket più lontani del suo vicinato.

## 3 Scelte di progetto

Di seguito vengono descritte le principali scelte di progetto. La rete è modellata da una serie di processi Erlang separati che rappresentano dei nodi Kademlia. Ciascun nodo ha un proprio stato interno e comunica con il resto dei nodi della rete tramite dei messaggi. Un nodo funziona grazie ad un loop principale che gli permette di rimanere in ascolto dei messaggi in arrivo e di reagire ad essi tramite l'esecuzione di funzioni dedicate.

### 3.1 Stato di un nodo

Le informazioni di stato di un nodo sono incapsulate all'interno di una mappa chiave-valore. Queste includono sia variabili dinamiche che vengono aggiornate durante l'operatività del nodo (e.g. buckets), sia parametri globali selezionabili dall'utente in fase di avvio della rete (e.g.  $k$  e  $\alpha$ ). Si è scelto di usare una mappa perché permette di incapsulare meglio le informazioni e passarle alle varie funzioni come un unico argomento, migliorando la leggibilità.

I parametri globali di rete configurabili sono:

- **k\_param**: è il valore di replicazione. Controlla il numero massimo di contatti all'interno di ciascun bucket e il numero di nodi più vicini restituiti dalla procedura di node lookup;
- **alpha\_param**: è il valore di parallelismo. Controlla il numero di nodi che vengono contattati simultaneamente durante una procedura di node lookup;
- **id\_byte\_length**: la lunghezza degli ID dei nodi e delle chiavi espressa in byte. Il numero di bit degli ID generati controlla il numero di bucket presenti all'interno della tabella di routing di ciascun nodo (e.g. 20 byte comporta 160 bit, quindi 160 bucket);
- **republish\_interval**, **expiration\_interval** e **refresh\_interval**: sono rispettivamente l'intervallo di tempo dopo il quale fare republish di un valore, eliminare un entrata scaduta ed effettuare un refresh dei bucket;
- **timeout\_interval**: intervallo che controlla dopo quanto tempo dichiarare il timeout di una RPC inviata.

Nello stato sono inoltre presenti delle strutture dati necessarie al funzionamento di un nodo, che includono:

- **local\_ID**: è un ID casuale assegnato al nodo durante il suo avvio, avente dimensione `id_byte_length`;
- **buckets**: è una lista bidimensionale che rappresenta la tabella di routing del nodo. Ciascuna lista all'interno è un k-bucket. Ogni k-bucket contiene fino a  $k$  nodi, ciascuno rappresentato da una tripla `<NodeName, NodeID, NodePid>`. Il `NodeName` è un nome assegnato al nodo, utile per il logging; `NodeID` è l'identificatore assegnato al nodo; infine, `NodePid` è il pid del processo Erlang corrispondente al nodo, utilizzabile per contattarlo;
- **storage**: è una hash map che contiene delle coppie `<Key, Entry>` ricevute da altri nodi tramite messaggi `STORE`. La chiave `Key` è l'hash del valore. L'elemento `Entry` è una quadrupla `<Value, Expiry, Republish, Owner>`. `Value` è il valore; `Expiry` è il tempo di scadenza dopo il quale questa entrata dovrà essere eliminata dal nodo; `Republish` è il tempo di republish dopo il quale il valore dovrà essere ripubblicato nella rete; infine, `Owner` rappresenta il nodo che per primo ha pubblicato questo valore sulla rete, informazione utile agli altri nodi per poter aggiornare i tempi di scadenza dei valori ricevuti che sono già nel loro storage (solo i republish da parte del nodo originale devono poter aggiornare il tempo di scadenza di un valore).

Oltre ai parametri globali e alle strutture interne di un nodo, la mappa di stato contiene anche delle variabili per gestire il lookup. Per svolgere un lookup in maniera asincrona, bisogna mantenere nello stato interno di un nodo anche delle variabili che tengano traccia delle informazioni di un processo di lookup in corso. Di seguito un elenco di queste variabili:

- **lookup\_status**: flag che indica se il nodo ha un processo di lookup in corso o meno;
- **is\_last\_query**: flag che indica se bisogna terminare il processo di lookup;
- **lookup\_mode**: specifica l'operazione in corso (e.g. store, find node, ecc.);
- **visited\_nodes**: lista di nodi visitati durante il processo di lookup;
- **nodes\_collected**: lista di nodi raccolti durante la procedura di node lookup. La lista è mantenuta ordinata in base alla distanza dal ID target. Quando termina un round di lookup, si controlla se in questa lista è presente un nodo più vicino del più vicino dei nodi in `visited_nodes`. In tal caso, si continua; altrimenti, se non ci sono nuovi nodi più vicini e non sono presenti altre richieste pendenti, si esegue un'ultima lookup sui  $k$  nodi più vicini raccolti e si imposta il flag di terminazione `is_last_query`. Al termine, si restituiscono i risultati in base all'operazione svolta (e.g. se la procedura era una `FIND_NODE`, si prendono come risultato i  $k$  nodi più vicini in `nodes_collected`, se era una `FIND_VALUE` si restituisce il valore trovato o `not_found`);
- **pending\_requests** e **timeout\_refs**: la prima è una lista che contiene le richieste pendenti inviate durante il processo di lookup. Fintanto che ci sono richieste pendenti, il processo non termina, in quanto bisogna ancora ricevere risposta da alcuni nodi contattati. La seconda invece è un mappa che contiene, per ogni richiesta pendente, una coppia `<Pid, Timer>`, dove `Pid` è il destinatario della richiesta e `Timer` è un timer che scatta quando non si riceve risposta da quel nodo entro l'intervallo di tempo `timeout_interval`. Il timeout fa scattare l'invio di un messaggio al nodo stesso, informandolo di quale richiesta ha superato il limite di tempo così che il nodo possa rimuovere la richiesta da `pending_requests`. Non appena `pending_requests` è vuota, il lookup può terminare.



### 3.2 Bootstrap di un nodo

I nodi di bootstrap sono speciali nodi che hanno lo stesso comportamento dei nodi normali ma possono essere scelti dai nuovi nodi per entrare a far parte della rete. I nodi di bootstrap sono creati all'avvio della rete e vengono monitorati da un supervisor, il quale li riavvia in caso di guasti inaspettati. In questo modo, la rete rimane sempre accessibile dai nuovi nodi che vogliono entrare, anche in presenza di guasti da parte di alcuni nodi di bootstrap.

I nodi di bootstrap sono salvati all'interno di una tabella che può essere acceduta globalmente. Un nuovo nodo che entra seleziona casualmente un nodo di bootstrap tra quelli presenti in quel momento nella tabella. Se non esiste ancora un nodo di bootstrap, allora il nodo stesso diventa un nodo di bootstrap.

### 3.3 Republish e Refresh

Sono state implementate anche le logiche per gestire il republish periodico dei valori sulla rete e il refresh periodico dei bucket di ogni nodo. Entrambe le procedure fanno uso di processi separati dal processo principale del nodo. Questo è stato fatto per due motivi: 1) per parallelizzare le operazioni e quindi migliorare l'efficienza e 2) per semplificare la gestione delle lookup di un nodo. Infatti, per quanto riguarda il secondo punto, senza utilizzare dei processi separati si sarebbero dovute gestire tutte le lookup svolte nel processo principale, dovendo tenere traccia dello stato di ciascuna in una mappa o qualche altra struttura. Tramite la gestione con processi separati, invece, per ogni lookup che viene lanciata viene creato un processo dedicato che si occupa di gestire solo quel singolo lookup e che invierà un risultato al processo principale al termine della procedura tramite un messaggio.

## 4 Implementazione

### 4.1 Struttura del progetto

In questa sezione si descrive l'implementazione di ciascuna delle scelte di progetto descritte nella sezione precedente, accompagnando le spiegazioni ai corrispondenti pezzi di codice sorgente.

Il progetto è suddiviso in diversi moduli. Di seguito un elenco e descrizione di ciascun modulo:

- **node.erl**: contiene le logiche di funzionamento di un nodo Kademlia. Include le funzioni per avviare la rete, le funzioni per creare/inizializzare i nodi e la funzione che contiene il loop principale di funzionamento di un nodo;
- **bootstrap\_node.erl**: contiene la funzione di start del worker **bootstrap**, in accordo alle specifiche del pattern Supervisor OTP;
- **bootstrap\_sup.erl**: contiene la funzione di start del supervisor che gestisce i vari worker **bootstrap\_node**;
- **utils.erl**: contiene funzioni ausiliarie utilizzate negli altri moduli, come le funzioni per calcolare la distanza XOR, per generare gli ID e stampare a schermo lo stato di un nodo;
- **test.erl**: modulo per testare le funzionalità dell'implementazione. Richiama le funzioni create negli altri moduli per eseguire misurazioni e altri test;

- `log.erl`: modulo con funzioni per il logging, che permette di stampare delle stringhe con diversi livelli di importanza, su console e su file.

Per compilare agevolmente tutti i moduli del progetto è presente un `Emakefile`, che permette di compilare tutti i sorgenti (`.erl`) eseguendo un unico comando nella radice del progetto, andando ad inserire tutti i file compilati (`.beam`) nella cartella `ebin`:

```
make:all([outdir, './ebin'])).
```

## 4.2 Avvio della rete

Prima di poter aggiungere un nodo, bisogna creare e inizializzare la rete. Ciò può essere fatto tramite la funzione `node:start_network/1`, che prende in input la mappa con i parametri globali di configurazione descritta sopra. La funzione inizializza il logger. Crea la tabella ETS che conterrà i parametri di rete globali e vi inserisce le opzioni passate come argomento della funzione, rendendole così accessibili a qualunque nodo. Infine, viene inizializzato il supervisor dei nodi di bootstrap e creata la tabella ETS che conterrà questi ultimi.

Listing 1: `node:start_network/1`

```
%% Starts the Kademlia network with custom options.
start_network(Options) ->
    IdByteLength = maps:get(id_byte_length, Options, 20),

    % Set logger
    LogLevel = maps:get(log_level, Options, info),
    log:set_level_global(LogLevel),
    log:info("~n~nSTARTING NETWORK~n"),
    log:info("ID byte length = ~p", [IdByteLength]),

    % Create ETS table for shared options
    case ets:info(network_options) of
        undefined ->
            % Table doesn't exist, create it
            ets:new(network_options, [named_table, public, {keypos, 1}]);
        _ ->
            ets:delete_all_objects(network_options)
    end,
    ets:insert(network_options, {global_options, Options}),

    % Start bootstrap supervisor
    SupPid = try
        {ok, Pid} = bootstrap_sup:start_link(),
        Pid
    catch
        error:{badmatch, {error, {already_started, AlreadyStartedPid}}} ->
            log:warn("bootstrap_sup already started"),
            AlreadyStartedPid;
        _:Reason ->
            error_logger:error_msg("Error during startup of bootstrap_sup: ~p",
                [Reason]),
            erlang:error(Reason)
    end,
```

```

% Create ETS table for bootstrap nodes
case ets:info(bootstrap_nodes) of
    undefined ->
        ets:new(bootstrap_nodes, [named_table, public, {keypos, 1}]),
        log:info("Table bootstrap_nodes created");
    _ ->
        log:info("Table bootstrap_nodes already exists")
end,
{ok, SupPid}.

```

### 4.3 Creazione e inizializzazione dei nodi

Dopo aver creato la rete, è possibile avviare nuovi nodi. Innanzitutto, bisogna avviare almeno un nodo di bootstrap. Ciò può essere fatto tramite la funzione `node:start_bootstrap_node/3`. Questa funzione crea una *child spec* per il worker, ovvero una specifica da passare al supervisor che gli dice come dovrà essere gestito questo worker. La *child spec* contiene il nome del worker, la funzione di avvio/riavvio del worker, il tipo di riavvio (e.g. `transient`) e il tempo di attesa tra un riavvio e l'altro. La policy di restart da parte del supervisor è impostata a `transient`, facendo sì che il worker venga riavviato solo in caso di terminazione anomala (motivazione di exit diversa da "normal").

Listing 2: `node:start_bootstrap_node/3`

```

%% Creates a new bootstrap node using the Supervisor OTP behaviour.
start_bootstrap_node(Name, ShellPid, IdLength) ->
    log:info("~p: starting..", [Name]),
    ChildSpec = {Name,
        {bootstrap_node, start_link, [Name, ShellPid, IdLength]},
        transient,
        5000,
        worker,
        [bootstrap_node]},
    supervisor:start_child({global, bootstrap_sup}, ChildSpec).

```

Nella *childspec* è presente il riferimento alla funzione di start del worker, che è la funzione `bootstrap_node:start_link/3`. Questa spawna un nuovo processo a cui viene passata la funzione `init`. La funzione privata `init/3` genera un ID casuale per il nodo e inserisce il nodo nella tabella dei nodi di bootstrap. Anche i nodi di bootstrap (ad eccezione del primo) hanno bisogno di un altro nodo di bootstrap per entrare, e questo viene scelto richiamando la funzione `node:choose_bootstrap/1`, che seleziona un nodo casuale dalla tabella ETS dei nodi di bootstrap precedentemente creati. Infine, si richiama la funzione `node:init_node` che contiene la logica di inizializzazione di un qualunque nodo Kademlia.

Listing 3: `bootstrap_node:start_link/3`

```

start_link(Name, ShellPid, IdLength) ->
    Pid = spawn_link(fun() -> init(Name, ShellPid, IdLength) end),
    log:info("Spawned bootstrap node: ~p~n", [{Name, Pid}]),
    {ok, Pid}.

init(Name, ShellPid, IdLength) ->
    ID = utils:generate_node_id(IdLength),

```

```

% Save the bootstrap info in an ETS table
ets:insert(bootstrap_nodes, {Name, ID, self()}),
log:info("Saved bootstrap node: ~p~n", [Name]),

% Choose a bootstrap node for this bootstrap node
BootstrapNode = node:choose_bootstrap(Name),
log:info("Chosen bootstrap node: ~p~n", [BootstrapNode]),
node:init_node(ID, BootstrapNode, Name, ShellPid),
ok.

```

Per creare un nodo normale invece si utilizza la funzione `node:start_simple_node/2`. Anche qui si seleziona un nodo di bootstrap, si assegna un ID casuale al nodo e si spawna un nuovo processo passandogli la funzione `node:init_node`.

Listing 4: `node:start_simple_node/2`

```

%% Creates a new Kademlia node with a random node ID.
start_simple_node(LocalName, IdLength) ->
  log:info("~p: starting..", [LocalName]),
  ShellPid = self(),

  % Select a bootstrap node from available bootstrap nodes
  choose_bootstrap(LocalName),
  {BootstrapName, _BootstrapId, BootstrapPid} = BootstrapNode,
  log:info("~p: selected Bootstrap node ~p (~p)", [LocalName,
    BootstrapName, BootstrapPid]),

  % Generate random binary Kademlia ID
  NodeId = utils:generate_node_id(IdLength),

  NodePid = spawn(fun() -> init_node(NodeId, BootstrapNode, LocalName,
    ShellPid) end),
  log:info("~p (~p): spawned node", [LocalName, NodePid]),
  {ok, {LocalName, NodeId, NodePid}}.

choose_bootstrap(OwnName) ->
  % Select a random bootstrap node (expect itself)
  AllNodes = ets:match_object(bootstrap_nodes, {'_', '_', '_'}),
  OtherNodes = [{Name, ID, Pid} || {Name, ID, Pid} <- AllNodes,
    Name /= OwnName],
  case OtherNodes of
    [] ->
      undefined; % First bootstrap node
    _ ->
      {Name, ID, Pid} = lists:nth(rand:uniform(length(OtherNodes)),
        OtherNodes),
      {Name, ID, Pid}
  end.

```

La funzione `init_node` del modulo `nodo.erl` consente di inizializzare lo stato di un nodo e far partire il suo loop di funzionamento. Come prima cosa, si leggono i parametri globali di rete dalla tabella ETS. Poi si creano le strutture dedicate per gestire lo stato di questo nodo, tra cui la lista di bucket e la mappa di coppie chiave-valore, entrambe inizialmente vuote, e si unisce tutto in un'unica mappa di stato.

Vengono fatti partire due timer periodici: uno gestisce il refresh periodico dei bucket,

mentre l'altro controlla periodicamente la scadenza dei valori. Se un valore è scaduto, viene eliminato dalla mappa del nodo.

Nel case ci sono due rami: nel primo caso, se `BootstrapNode` è `undefined`, significa che questo è il primo nodo di bootstrap, quindi non si fa nient'altro. Nel secondo caso invece, si inserisce il nodo di bootstrap nel giusto bucket tramite `update_routing_table`, e si avvia una node lookup per l'ID del nodo stesso, in modo da trovare i suoi vicini e farsi conoscere. Dopo aver avviato la lookup, il nodo entra subito nel suo loop (`node_loop`), mettendosi in attesa di ricevere le prime risposte.

Listing 5: `node:init_node/4`

```
%% Initializes a spawned Kademlia node before calling the node loop.
init_node(ID, BootstrapNode, LocalName, ShellPid) ->
  Self = self(),
  log:info("~p (~p): node initialization", [LocalName, Self]),

  % Get global network options from ETS table
  [{global_options, NetworkOpts}] = ets:lookup(network_options,
                                                global_options),

  % Storage and buckets list creation
  Storage = #{},
  IdByteLength = maps:get(id_byte_length, NetworkOpts, 20),
  IdBitLength = IdByteLength * 8,
  Buckets = lists:duplicate(IdBitLength, []),

  % Lookup state variables
  VisitedNodes = [],
  NodesCollected = [],
  PendingRequests = [],
  LookupStatus = done,
  TimeoutRefs = #{},

  % Combine global options with with node specific state variables
  State = maps:merge(NetworkOpts, #{
    local_id => ID,
    self => self(),
    local_name => LocalName,
    buckets => Buckets,
    storage => Storage,
    visited_nodes => VisitedNodes,
    nodes_collected => NodesCollected,
    lookup_status => LookupStatus,
    pending_requests => PendingRequests,
    timeout_refs => TimeoutRefs
  }),

  % Start a timer that will send 'refresh_buckets' every refresh_interval
  timer:send_interval(RefreshInterval, self(), refresh_buckets),

  % Start a timer that will send 'check_expiration' every interval
  timer:send_interval(CheckExpirationinterval, self(), check_expiration),

  K = maps:get(k_param, State),
  Alpha = maps:get(alpha_param, State),
  LocalId = maps:get(local_id, State),
```

```

CheckExpirationinterval = maps:get(check_expiration_interval, State,
    60000),
case BootstrapNode of
    undefined -> % first bootstrap node
        log:info("~p (~p): first bootstrap", [LocalName, Self]),
        ShellPid ! {find_node_response, []},
        node_loop(State);
    _ ->
        % Insert bootstrap node in this node's bucket list
        {BootName, _BootId, BootPid} = BootstrapNode,
        log:debug("~p (~p): inserting bootstrap node ~p (~p)",
            [LocalName, Self, BootName, BootPid]),
        UpdatedBuckets = update_routing_table([BootstrapNode],
            LocalId, Buckets, K, LocalName, IdBitLength),
        utils:print_buckets_with_distance(LocalId, LocalName, Self,
            UpdatedBuckets),

        % Search nodes closer to local ID (self-lookup)
        log:debug("~p (~p): executing self lookup..", [LocalName, Self]),
        NewState = nodes_lookup(LocalId,
            State#{lookup_status => in_progress,
                lookup_mode => find_node,
                requester => ShellPid,
                buckets => UpdatedBuckets},
            Alpha),
        node_loop(NewState)
end.

```

#### 4.4 Funzionamento di un nodo

Nel modulo `nodo.erl` è presente la funzione `node_loop/1`, che contiene il loop principale di un nodo Kademlia. All'interno del loop il nodo si mette in attesa dei messaggi all'interno di un blocco `receive`. Di seguito, verranno descritti i principali messaggi gestiti nel blocco `receive` della funzione `node_loop` di ciascun nodo. Dopodiché, si parlerà di come viene implementata la logica centrale di `node lookup`.

Ciascun nodo accetta i quattro messaggi di protocollo:

- **PING**: quando riceve questo messaggio, il nodo invia la risposta (PONG) al mittente e inserisce ques'ultimo nella tabella di routing.

Listing 6: Gestore messaggio 'PING'

```

{'PING', FromNode} ->
    {FromName, _FromID, FromPid} = FromNode,
    log:info("~p (~p): PING message from ~p (~p)", [LocalName, Self,
        FromName, FromPid]),
    FromPid ! {'PONG', {LocalName, LocalId, Self}},
    node_loop(State);

```

- **STORE**: il valore ricevuto viene salvato nella mappa di coppie chiave-valore, si inserisce il mittente nel corretto bucket, e infine si invia un ACK al mittente, per notificarlo dell'avvenuto salvataggio.

Listing 7: Gestore messaggio 'STORE'

```

{'STORE', Data, Node, FromPid} ->
{NodeName, _NodeId, NodePid} = Node,
log:info("~p (~p): STORE message from ~p (~p)", [LocalName, Self,
  NodeName, NodePid]),

% Save pair inside the local storage
{Key, Entry} = Data,
UpdatedStorage = store_locally(Key, Entry, Storage, RepublishInterval),
NewBuckets = update_routing_table([Node], LocalId, Buckets, K,
  LocalName, IdBitLength, TimeoutInterval),

% Print new state
utils:print_buckets_with_distance(LocalId, LocalName, Self, NewBuckets,
  Key, debug),
utils:print_storage(LocalName, Self, UpdatedStorage),

% Send ACK back to sender
log:debug("~p (~p): sending STORE_ACK to ~p (~p)", [LocalName, Self,
  NodeName, NodePid]),
FromPid ! {'STORE_ACK', self()},

node_loop(State#{buckets => NewBuckets, storage => UpdatedStorage});

```

- **FIND\_NODE**: dopo aver aggiornato la tabella di routing con il contatto ricevuto, invia al mittente i  $k$  nodi più vicini al **TargetID** presi dalla tabella di routing.

Listing 8: Gestore messaggio 'FIND\_NODE'

```

{'FIND_NODE', TargetId, Node, FromPid} ->
{NodeName, _NodeId, NodePid} = Node,
log:info("~p (~p): FIND_NODE message from ~p (~p)", [LocalName, Self,
  NodeName, NodePid]),

% Add the sender node to the routing table
NewBuckets = update_routing_table([Node], LocalId, Buckets, K,
  LocalName, IdBitLength, TimeoutInterval),
utils:print_buckets_with_distance(LocalId, LocalName, Self,
  NewBuckets, TargetId, debug),

% Send back the K closest nodes to target ID from this node's buckets
ClosestNodes=get_closest_from_buckets(NewBuckets, TargetId, K, Node),
utils:print_nodes_with_distance(ClosestNodes,
  TargetId,
  io_lib:format("~p (~p) - FIND_NODE: return ~p closest",
    [LocalName, Self, length(ClosestNodes)])),
FromPid ! {'FIND_NODE_RESPONSE', {nodes, ClosestNodes}, TargetId,
  {LocalName, LocalId, Self}},

node_loop(State#{buckets => NewBuckets});

```

- **FIND\_VALUE**: ricerca il valore richiesto (Key) nel proprio storage. Se lo trova lo restituisce, altrimenti restituisce i  $k$  nodi più vicini al **TargetID** esattamente come la **FIND\_NODE**.

Listing 9: Gestore messaggio 'FIND\_VALUE'

```

{'FIND_VALUE', Key, Node, FromPid} ->
{NodeName, _NodeId, NodePid} = Node,
log:info("~p (~p): FIND_VALUE message from ~p (~p)", [LocalName,
  Self, NodeName, NodePid]),

% Add the sender node to the routing table
NewBuckets = update_routing_table([Node], LocalId, Buckets, K,
  LocalName, IdBitLength, TimeoutInterval),
utils:print_buckets_with_distance(LocalId, LocalName, Self,
  NewBuckets, Key, debug),

% Search locally for value. If not found, return closest nodes.
case get_entry_from_storage(Key, Storage, LocalName) of
{entry, Entry} when Entry /= not_found ->
  log:debug("~p (~p) - FIND_VALUE: sending value found locally",
    [LocalName, Self]),
  FromPid ! {'FIND_VALUE_RESPONSE', {value, Entry}, Key,
    {LocalName, LocalId, Self}};
{entry, not_found} ->
  ClosestNodes = get_closest_from_buckets(NewBuckets, Key, K, Node),
  log:debug("~p (~p) - FIND_VALUE: NOT found, returning ~p closest",
    [LocalName, Self, length(ClosestNodes)]),
  FromPid ! {'FIND_VALUE_RESPONSE', {nodes, ClosestNodes}, Key,
    {LocalName, LocalId, Self}}
end,
node_loop(State#{buckets => NewBuckets})

```

È necessaria un'interfaccia per poter comandare il nodo, e fargli eseguire delle azioni, come fare una lookup di un ID o avviare una procedura di store di un valore. Questa è implementata da ulteriori blocchi all'interno della `receive` del loop principale. Tutte le procedure, a eccezione della PING, invocano la funzione `node_lookup`, impostando un diverso valore per la variabile di stato `lookup_mode` (e.g. `lookup_mode=find_node`). Per brevità, viene riportata qui sotto solo la gestione del messaggio d'interfaccia `store_request`:

Listing 10: Gestore messaggio `store_request`

```

{store_request, Value, FromPid} ->
  log:info("~p: store_request from ~p", [LocalName, FromPid]),
  Key = utils:calculate_key(Value, IdByteLength),
  NewState = node_lookup(Key,
    State#{lookup_status => in_progress,
      lookup_mode => store,
      requester => FromPid,
      value_to_store => Value},
    Alpha),
  node_loop(NewState);

```

## 4.5 Logica di lookup

La funzione `node_lookup` e la funzione `handle_lookup_response` sono le funzioni centrali alla base dell'implementazione dei lookup Kademlia. La funzione `node_lookup` seleziona innanzitutto gli  $\alpha$  nodi più vicini al target ID. Se non ci sono nuovi nodi non ancora visitati, si entra nel primo ramo della `case`. Nel primo ramo, se il flag `is_last_query` è `true` e non



ci sono richieste pendenti, il lookup termina (messaggio `trigger_final_response`). Se ci sono ancora nodi da visitare si entra invece nel secondo ramo e si inviano delle query (find node o find value) ai nodi selezionati, si aggiornano i nodi visitati, le richieste pendenti e si inizializzano i timer per i timeout. Infine, si restituisce il nuovo stato per il lookup in corso.

Listing 11: `node:node_lookup/3`

```

%% Initiates or continues a node lookup process for a given target ID.
node_lookup(TargetID, State, Alpha) ->
    Buckets = maps:get(buckets, State),
    LocalName = maps:get(local_name, State, unknown),
    LocalId = maps:get(local_id, State),
    VisitedNodes = maps:get(visited_nodes, State),
    NodesCollected = maps:get(nodes_collected, State),
    LookupMode = maps:get(lookup_mode, State),
    Hops = maps:get(num_hops, State, 0),
    Main = maps:get(self, State),
    Self = self(),

    {ClosestNodes, NewNodesCollected} = select_closest_nodes(TargetID, Alpha,
        Buckets, NodesCollected, VisitedNodes, Main, LocalName, LocalId),
    utils:print_nodes_with_distance(ClosestNodes,
        TargetID,
        io_lib:format("~p (~p) - node_lookup - ClosestNodes: ",
            [LocalName, Main])),

    NewState = case ClosestNodes of
        [] ->
            log:debug("~p (~p): No more nodes to lookup", [LocalName, Main]),

            % Check if this was the last query and no pending requests
            IsLastQuery = maps:get(is_last_query, State, false),
            Pending = maps:get(pending_requests, State, []),
            case {IsLastQuery, Pending} of
                {true, []} ->
                    Self ! trigger_final_response,
                    State;
                _ ->
                    State
            end;
        _ ->
            % Query closest nodes
            {TimeoutRefs, Pending} = case LookupMode of
                find_value ->
                    find_query(ClosestNodes, TargetID, find_value, State);
                _ ->
                    find_query(ClosestNodes, TargetID, find_node, State)
            end,
            NewVisitedNodes = VisitedNodes ++ ClosestNodes,
            utils:print_nodes_with_distance(NewVisitedNodes,
                TargetID,
                io_lib:format("~p (~p) - node_lookup - VisitedNodes: ",
                    [LocalName, Main])),
            NewHops = Hops + 1,
            State#{

```

```

        visited_nodes => NewVisitedNodes,
        timeout_refs => TimeoutRefs,
        pending_requests => Pending,
        nodes_collected => NewNodesCollected,
        num_hops => NewHops
    }
end,
NewState.

```

A questo punto il nodo rimane in attesa nel loop principale di ricevere un messaggio di risposta da parte dei nodi contattati. Considerando il caso della procedura `find node`, di seguito è riportato il ramo del blocco `receive` che si occupa di gestire le risposte da parte di un round di lookup. Come si può vedere, se il lookup è terminato (`lookup_done=done`), si ignora il messaggio; altrimenti, se il lookup è ancora in corso (`lookup_done=in_progress`), si procede chiamando la funzione `handle_lookup_response`.

Listing 12: Gestore messaggio 'FIND\_NODE\_RESPONSE'

```

{'FIND_NODE_RESPONSE', {nodes, ReturnedNodes}, TargetID, FromNode} ->
{FromName, _FromId, FromPid} = FromNode,
case maps:get(lookup_status, State) of
    in_progress ->
        log:debug("~p (~p): received a FIND_NODE_RESPONSE from ~p (~p)",
            [LocalName, Self, FromName, FromPid]),
        handle_lookup_response(ReturnedNodes, TargetID, FromNode, State);
    done ->
        % Lookup already terminated
        log:debug("~p (~p): ignoring FIND_NODE_RESPONSE from ~p (~p)",
            [LocalName, Self, FromName, FromPid]),
        node_loop(State)
end;

```

Il funzionamento della `handle_lookup_response` è il seguente:

1. rimuove il nodo che ha inviato la risposta dalla lista `pending_requests`;
2. aggiorna la tabella di routing con i nuovi nodi ricevuti (`update_routing_table()`);
3. aggiunge i nuovi nodi alla lista `nodes_collected`, ordinandola per distanza crescente dal ID target e rimuovendo i duplicati;
4. controlla se tra i nodi in `nodes_collected` esiste almeno un nodo più vicino del più vicino dei nodi in `visited_nodes`. Se esiste, procede con una nuova lookup; se invece non è stato trovato un nuovo nodo più vicino, i casi sono due: ci sono ancora richieste pendenti, e quindi potenziali nodi più vicini non ancora ricevuti, pertanto si attende; oppure, non ci sono richieste pendenti. In quest'ultimo caso, si effettua un'ultima lookup per  $k$  nodi, invece che  $\alpha$ , impostando anche il flag `is_last_query` a `true` nello stato di lookup, in modo che nella successiva chiamata a `handle_lookup_response`, la lookup termini invece di continuare all'infinito.

Listing 13: `handle_lookup_response/4`

```

%% Handles a lookup response, updating the state accordingly.
handle_lookup_response(ReturnedNodes, TargetID, FromNode, State) ->
    LocalName = maps:get(local_name, State),
    LocalId = maps:get(local_id, State),

```

```

IdByteLength = maps:get(id_byte_length, State, 20),
IdBitLength = IdByteLength * 8,
Buckets = maps:get(buckets, State),
K = maps:get(k_param, State),
Alpha = maps:get(alpha_param, State),
TimeoutInterval = maps:get(timeout_interval, State),
PendingRequests = maps:get(pending_requests, State),
TimeoutRefs = maps:get(timeout_refs, State),
NodesCollected = maps:get(nodes_collected, State),
VisitedNodes = maps:get(visited_nodes, State),
LookupMode = maps:get(lookup_mode, State),
IsLastQuery = maps:get(is_last_query, State, false),
Main = maps:get(self, State),
{FromName, _, FromPid} = FromNode,

% Remove received node from pending requests
NewPending = lists:delete(FromNode, PendingRequests),
log:debug("~p (~p): removed ~p (~p) from pending requests",
  [LocalName, Main, FromName, FromPid]),
utils:print_nodes(NewPending,
  io_lib:format("~p (~p) - NewPending: ", [LocalName, Main])),

% Cancel timer
NewTimeouts = case maps:find(FromPid, TimeoutRefs) of
  {ok, TimeoutRef} ->
    erlang:cancel_timer(TimeoutRef),
    maps:remove(FromPid, TimeoutRefs);
  _ ->
    TimeoutRefs
end,

% Update routing table
FilteredNodes = [Node || {_, NodeId, _} = Node <- ReturnedNodes,
  NodeId /= LocalId], % Remove myself
NewBuckets = update_routing_table(FilteredNodes, LocalId, Buckets, K,
  LocalName, IdBitLength, TimeoutInterval),
utils:print_buckets_with_distance(LocalId, LocalName, Main, NewBuckets,
  TargetID, debug),

% Update collected nodes
NewNodesCollected = utils:order_unique(NodesCollected ++ ReturnedNodes,
  TargetID),
utils:print_nodes_with_distance(NewNodesCollected,
  TargetID,
  io_lib:format("~p (~p) - node_lookup - NewNodesCollected: ",
    [LocalName, Main])),
NewState = State#{
  buckets => NewBuckets,
  nodes_collected => NewNodesCollected,
  pending_requests => NewPending,
  timeout_refs => NewTimeouts
},

% Check if there are closer nodes
case checkCloserNodes(FilteredNodes, VisitedNodes, TargetID) of
  true ->

```

```

% Continue with lookup of Alpha closer nodes not yet visited
log:debug("~p (~p): CONTINUE lookup - still some closer nodes",
  [LocalName, Main]),
UpdatedState = node_lookup(TargetID, NewState, Alpha),
case LookupMode of
  republish -> subprocess_loop(UpdatedState);
  refresh -> subprocess_loop(UpdatedState);
  _ -> node_loop(UpdatedState)
end;
false ->
  log:debug("~p (~p): NO CLOSER NODES to lookup", [LocalName, Main]),

  % Check if there are pending requests
  case length(NewPending) of
    0 ->
      % No pending requests
      case IsLastQuery of
        false ->
          % Perform one last query for K closest nodes not yet visited
          log:debug("~p (~p): LAST lookup for K nodes not visited",
            [LocalName, Main]),
          UpdatedState = node_lookup(
            TargetID,
            NewState#{is_last_query => true},
            K),
          case LookupMode of
            republish -> subprocess_loop(UpdatedState);
            refresh -> subprocess_loop(UpdatedState);
            _ -> node_loop(UpdatedState)
          end;
          true ->
            finalize_lookup(NewState)
          end;
        ->
          % There are still pending requests
          log:debug("~p (~p): CONTINUE - still some pending requests",
            [LocalName, Main]),
          % utils:print_nodes(NewPending,
          %   io_lib:format("~p (~p) - NewPending: ", [LocalName, Main])),
          case LookupMode of
            republish -> subprocess_loop(NewState);
            refresh -> subprocess_loop(NewState);
            _ -> node_loop(NewState)
          end
        end
      end
    end.

```

Prima o poi nella `handle_lookup_response`, dopo aver impostato il flag di terminazione `is_last_query`, verrà chiamata la funzione `finalize_lookup`, che restituisce semplicemente i risultati al chiamante sulla base dell'operazione in svolgimento e ripulisce lo stato di lookup del nodo.

## 4.6 Republish dei valori

La logica di republish controlla periodicamente se sono presenti dei valori che hanno superato il periodo di ripubblicazione, calcolato sulla base del parametro `republish_interval`. Questo controllo viene fatto tramite la funzione `node:calculate_next_check_time`, che calcola il prossimo momento di republish sulla base delle entrate nello `storage`. Questo valore calcolato viene quindi impostato come timeout della receive del loop principale. Quando il timeout scatta, il nodo invia a se stesso il messaggio `republish_values`. Il messaggio ha l'effetto di eseguire la funzione `node:handle_republish`, che gestisce la logica di republish tramite un processo separato, mentre il nodo ritorna al proprio funzionamento normale.

Listing 14: Logica di republish di un valore

```
%% Manages the main loop of a Kademlia node, handling all incoming messages
%% and state transitions.
node_loop(State) ->

    ...

    % calculate next republish time
    NextRepublishTime = calculate_next_check_time(Storage,
    RepublishInterval),
    CurrentTime = erlang:system_time(millisecond),
    Timeout = max(0, NextRepublishTime - CurrentTime),

    receive

        ...

        {republish_values, FromPid} ->
            NewState = State#{requester => FromPid},
            handle_republish(NewState),
            node_loop(NewState);

        ...

    after Timeout ->
        log:important("~p (~p): value republish timeout", [LocalName, Self]),
        Self ! {republish_values, Self},
        node_loop(State)
    end.
```

La funzione `handle_republish` funziona in questo modo: innanzitutto, trova i valori che devono essere ripubblicati tramite `find_values_to_republish` (ovvero le entrate il cui campo `Republish` ha superato `CurrentTime`), poi resetta i tempi di republish per questi valori nel proprio storage. Infine, per ciascun valore da ripubblicare, fa la spawn di un nuovo processo, facendo partire al suo interno una lookup di tipo "republish" ed entrando in `subprocess_loop`. Se il nodo che sta facendo la republish è anche il possessore originale di quel valore, allora viene anche resettato il tempo di scadenza `Expiry`, in modo da informare gli altri nodi che il valore è ancora valido e quindi non è da eliminare.

Listing 15: `node:handle_republish/1`

```
%% Handles the republishing of values that are due for refresh.
handle_republish(State) ->
```

```

LocalName = maps:get(local_name, State),
LocalId = maps:get(local_id, State),
Storage = maps:get(storage, State),
RepublishInterval = maps:get(republish_interval, State),
ExpirationInterval = maps:get(expiration_interval, State),
Alpha = maps:get(alpha_param, State),
Self = self(),

% Find values to republish
ValuesToRepublish = find_values_to_republish(Storage, LocalName),

% Update republish times in storage
CurrentTime = erlang:system_time(milliseconds),
NewRepublishTime = CurrentTime + RepublishInterval,
UpdatedStorage = lists:foldl(
    fun({Key, Value, Expiry, Owner}, AccStorage) ->
        maps:put(Key, {Value, Expiry, NewRepublishTime, Owner},
            AccStorage)
    end,
    Storage,
    ValuesToRepublish
),
utils:print_storage(LocalName, Self, UpdatedStorage),

% Spawn separate lookup process for each value to republish
lists:foreach(
    fun({Key, Value, Expiry, Owner}) ->
        % If the value republished is mine, update also expiration time
        {_OwnerName, OwnerID, _OwnerPid} = Owner,
        NewExpiry = case OwnerID == LocalId of
            true -> CurrentTime + ExpirationInterval;
            false -> Expiry
        end,
        Entry = {Key, {Value, NewExpiry, Owner}},
        spawn(fun() ->
            LookupState = State#{
                lookup_status => in_progress,
                lookup_mode => republish,
                main_process => Self,
                value_to_store => Entry
            },
            NewState = node_lookup(Key, LookupState, Alpha),
            subprocess_loop(NewState)
        end)
    end,
    ValuesToRepublish
),
node_loop(State#{storage => UpdatedStorage}).

```

Infine, questa è la funzione `subprocess_loop`, che contiene la logica di gestione di una singola lookup (di republish o di refresh). Se il lookup è terminato (`lookup_status=done`), allora il sottoprocesso termina tramite `exit(normal)`; altrimenti, si mette in attesa di risposte da parte dei nodi contattati. Il procedimento è esattamente lo stesso di una normale operazione di lookup, già descritta precedentemente.

Listing 16: `node:sub_process_loop/1`

```

%% Manages the subprocess loop for parallel lookup operations
subprocess_loop(State) ->
    LocalName = maps:get(local_name, State),
    LookupStatus = maps:get(lookup_status, State),
    LookupMode = maps:get(lookup_mode, State),
    Main = maps:get(self, State),

    case LookupStatus of
        % If lookup has terminated, kill this lookup process
        done -> exit(normal);
        _ ->
            receive
                {'FIND_NODE_RESPONSE', {nodes, Nodes}, TargetID, FromNode} ->
                    {FromName, _FromId, FromPid} = FromNode,
                    log:debug("~p (~p) - ~p: FIND_NODE_RESPONSE from ~p (~p)",
                        [LocalName, Main, LookupMode, FromName, FromPid]),
                    handle_lookup_response(Nodes, TargetID, FromNode, State);

                % Message received when a pending request timeout happens
                {timeout, _Ref, NodePid} ->
                    log:debug("~p (~p) - ~p: lookup TIMEOUT for the node ~p",
                        [LocalName, Main, LookupMode, NodePid]),
                    NewState = handle_timeout(NodePid, State),
                    subprocess_loop(NewState);

                trigger_final_response ->
                    log:debug("~p (~p) - ~p: lookup finalization",
                        [LocalName, Main, LookupMode]),
                    finalize_lookup(State)
            end
    end.

```

#### 4.7 Refresh dei bucket

Una gestione simile alla republish dei valori è stata sviluppata per la refresh dei bucket. Nella funzione di inizializzazione di un nodo `init_node`, viene fatto partire un timer periodico tramite `timer:send_interval`, che invia ogni `RefreshInterval` un messaggio `refresh_buckets` al nodo corrente. Quando il messaggio è ricevuto, viene inserito nello stato il numero di buckets da refreshare e chiamata la funzione `handle_refresh`.

Listing 17: Logica di refresh dei bucket

```

%% Initializes a spawned Kademlia node before calling the node loop.
init_node(ID, BootstrapNode, LocalName, ShellPid) ->

    ...

    % Start a timer that will send 'refresh_buckets' every refresh_interval
    timer:send_interval(RefreshInterval, Self, {refresh_buckets, Self}),

    ...

%% Manages the main loop of a Kademlia node, handling all incoming messages
%% and state transitions.
node_loop(State) ->

```

```

...

{refresh_buckets, FromPid} ->
    NewState = State#{requester => FromPid, to_refresh => length(Buckets)},
    handle_refresh(NewState),
    node_loop(NewState);

...

```

La funzione `handle_refresh` fa la spawn di un nuovo processo per ciascuno degli indici della lista di bucket da refreshare. Per ciascun nuovo processo si fa partire una lookup con modalità `refresh` e si entra nella funzione di loop `subprocess_loop`, che è esattamente la stessa della procedura di republish.

Listing 18: `node:handle_refresh/1`

```

%% Handles the refreshing of routing table buckets.
handle_refresh(State) ->
    LocalId = maps:get(local_id, State),
    Buckets = maps:get(buckets, State),
    LocalName = maps:get(local_name, State),
    IdByteLength = maps:get(id_byte_length, State, 20),
    Alpha = maps:get(alpha_param, State),
    Self = self(),
    MaxIndex = length(Buckets),

    Indexes = lists:seq(0, MaxIndex - 1),

    % Spawn a separate lookup process for each bucket to refresh
    lists:foreach(fun(Index) ->
        spawn(fun() ->
            % Generate random ID for selected bucket
            log:important("~p (~p): REFRESHING bucket ~p",
                [LocalName, Self, Index]),
            TargetId = utils:generate_random_id_in_bucket(LocalId, Index,
                IdByteLength),

            % Initiate lookup loop
            LookupState = State#{
                lookup_status => in_progress,
                lookup_mode => refresh,
                main_process => Self
            },
            NewState = node_lookup(TargetId, LookupState, Alpha),
            subprocess_loop(NewState)
        end)
    end, Indexes),
    node_loop(State).

```

## 4.8 Funzioni ausiliarie

In questa sezione vengono descritte alcuni dei dettagli implementativi più di basso livello, riguardanti le funzioni di generazione e comparazione degli ID Kademlia presenti all'interno del modulo `utils.erl`.



**xor\_distance/2** La funzione `xor_distance/2` calcola lo XOR tra due ID binari, usando l'operatore Erlang `bxor`. La funzione `xor_distance_integer/2` fa la stessa cosa, ma decodifica il risultato in un intero prima di restituirlo. È usata per implementare la metrica di distanza Kademlia.

Listing 19: `utils:xor_distance/2`

```
% Calculates the XOR distance between two binary IDs of equal length.
xor_distance(A, B) when byte_size(A) == byte_size(B) ->
    list_to_binary(
        lists:map(fun({X, Y}) -> X bxor Y end,
            lists:zip(binary_to_list(A), binary_to_list(B))));
xor_distance(A, B) ->
    error({invalid_binary_lengths, byte_size(A), byte_size(B)}).

%% Calculates the XOR distance between two binary IDs and returns it as an
%% integer.
xor_distance_integer(A, B) ->
    BinaryXor = xor_distance(A, B),
    binary:decode_unsigned(BinaryXor).
```

**generate\_node\_id/1** Questa funzione genera un ID casuale di lunghezza `NumBytes`. Per farlo utilizza la funzione `strong_rand_bytes` del modulo `crypto`. Viene usata per assegnare degli ID casuali ai nodi entranti.

Listing 20: `utils:generate_node_id/1`

```
% Generates a random node ID with the specified number of bytes.
generate_node_id(NumBytes) when is_integer(NumBytes), NumBytes > 0 ->
    crypto:strong_rand_bytes(NumBytes).
```

**generate\_random\_id\_in\_bucket/3** Questa funzione permette di generare un ID casuale della giusta lunghezza che ricada all'interno di un bucket. Viene usata durante la refresh della tabella di routing di un nodo. Dato un indice di bucket, l'obiettivo della funzione è quello di creare un nuovo ID che abbia i bit prima dell'indice uguali all'ID locale (prefisso in comune) e i bit dopo l'indice totalmente casuali. L'algoritmo alla base è il seguente:

1. crea un ID casuale della lunghezza specificata (`RandomId`);
2. calcola la posizione del bit discriminante (`BitPos`) e del corrispondente byte (`BytePos`);
3. estrae il byte in cui ricade l'indice del bucket dall'ID locale (`LocalByte`) e da quello casuale (`RandomByte`);
4. crea un nuovo byte (`NewByte`) che contiene i bit di `LocalByte` fino a `BitPos`, il bit in `BitPos` invertito e i bit dopo `BitPos` presi da `RandomByte`. Per farlo crea due maschere, una per preservare i bit più significativi del byte estratto e una per il bit da invertire;
5. Combina i byte di `LocalId` fino a `BytePos`, il nuovo byte e i byte di `RandomId` dopo `BytePos`, a formare il nuovo ID.

Listing 21: `utils:generate_random_id_in_bucket/3`

```

%% Generates a random ID that would fall into a specific bucket relative to
%% a local ID.
generate_random_id_in_bucket(LocalId, BucketIndex, IdByteLength) ->
    % Create an ID that is formed by concatenating a prefix equal to
    % LocalID (up to BucketIndex), the inverted BucketIndex bit, and the
    % rest from RandomID

    % Generate random binary ID of the specified length
    RandomId = crypto:strong_rand_bytes(IdByteLength),

    % Calculate the position of the byte in LocalID
    BytePos = BucketIndex div 8,

    % Calculate the bit offset in the byte from the right,
    BitPos = 7 - (BucketIndex rem 8),

    % Converts the binaries to lists of bytes
    LocalBytes = binary_to_list(LocalId),
    RandomBytes = binary_to_list(RandomId),

    % Prefix equal to LocalID (before BucketIndex)
    NewIdPrefixBytes = lists:sublist(LocalBytes, BytePos),

    % Byte in BytePos (the bucket index starts from zero, so it adds 1)
    LocalByte = lists:nth(BytePos + 1, LocalBytes),
    RandomByte = lists:nth(BytePos + 1, RandomBytes),

    % Mask for bits that need to be equal to local ID
    Mask = (255 bsl (BitPos + 1)),

    % Mask for bit that needs to be different
    BitMask = (1 bsl BitPos),

    % New byte is equal to:
    %   bits up to bit pos equal to local byte +
    %   inverted bit +
    %   bits after bitPos equal to random byte
    NewByte = (LocalByte band Mask) bxor (BitMask) bor
              (RandomByte band (255 - Mask - BitMask)) band 255,

    % Remaining bytes are random
    NewIdSuffixBytes = lists:nthtail(BytePos + 1, RandomBytes),

    % Concatenates the parts
    GeneratedID = list_to_binary(NewIdPrefixBytes
                                ++ [NewByte]
                                ++ NewIdSuffixBytes),

    GeneratedID.

```

**find\_bucket\_index/3** Funzione che calcola l'indice del bucket in cui inserire un ID remoto, comparandolo con l'ID locale. Innanzitutto, calcola lo XOR tramite la funzione `xor_distance`. Poi, facendo uso della funzione `find_msb_set/2`, trova ricorsivamente il primo bit a 1 dello XOR partendo da sinistra, che rappresenta l'indice del bit più

significativo diverso tra i due ID.

Listing 22: `utils:find_bucket_index/3`

```
%% Finds the bucket index (distance) between a local ID and remote ID.
find_bucket_index(LocalId, RemoteId, IdBitLength) ->
    % Calculate XOR of the two IDs
    Distance = xor_distance(LocalId, RemoteId),
    % Find MSB
    find_msb_set(Distance, IdBitLength).

%% Helper function that finds the most significant bit set in a XOR distance.
find_msb_set(Distance, IdBitLength) when is_binary(Distance) ->
    IntDistance = binary:decode_unsigned(Distance),
    find_msb_set(IntDistance, 0, IdBitLength).

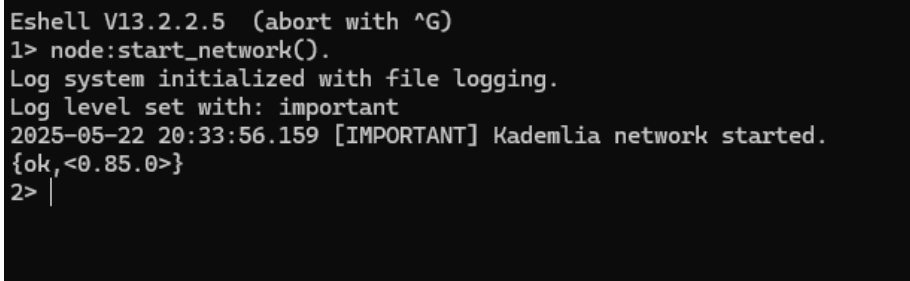
% Base case: zero distance -> no difference between the two IDs
find_msb_set(0, _, _) -> 0;
% Recursive function that iterates on the bits
find_msb_set(Distance, BitPos, IdBitLength) ->
    BitMask = 1 bsl ((IdBitLength-1) - BitPos),
    case Distance band BitMask of
        0 ->
            % Current bit is zero -> continue to next bit
            find_msb_set(Distance, BitPos + 1, IdBitLength);
        _ ->
            % Current bit is one -> found MSB
            BitPos
    end.
```

## 5 Testing

Per dimostrare il funzionamento dell'implementazione sono stati svolti diversi test di validazione sulle principali funzionalità implementate. Di seguito ne sono riportati alcuni.

### 5.1 Creazione rete e nodi

#### Avvio della rete



```
Eshell V13.2.2.5 (abort with ^G)
1> node:start_network().
Log system initialized with file logging.
Log level set with: important
2025-05-22 20:33:56.159 [IMPORTANT] Kademlia network started.
{ok,<0.85.0>}
2> |
```

Figure 4: Esecuzione e output della funzione `start_network` per inizializzare la rete.

#### Aggiunta di due nodi alla rete

```

Eshell V13.2.2.5 (abort with ^G)
1> node:start_network().
Log system initialized with file logging.
Log level set with: important
2025-05-21 08:55:00.648 [IMPORTANT] Kademlia network started.
{ok,<0.85.0>}
2>
2> node:start_bootstrap_node(bootstrap1, self()).
{ok,<0.87.0>}
3>
3> node:start_simple_node(node1).
{ok,{node1,<<9,95,25,1,127>>,<0.90.0>}}
4>
4> |

```

Figure 5: Dopo la creazione della rete, sono stati aggiunti due nodi, un nodo di bootstrap iniziale e un nodo semplice, tramite esecuzione delle funzioni `start_bootstrap_node` e `start_simple_node`.

### Esecuzione della funzione `test:prepare_network/2`

```

Eshell V13.2.2.5 (abort with ^G)
1> test:prepare_network(2,10).
Log system initialized with file logging.
2025-05-21 08:56:38.974 [INFO] Starting the network with 2 BOOTSTRAP nodes ----
Log system initialized with file logging.
Log level set with: important
2025-05-21 08:56:38.979 [IMPORTANT] Kademlia network started.
2025-05-21 08:56:39.605 [IMPORTANT]
REACHABLE NODES: [{node_1,<0.90.0>},
                  {node_2,<0.107.0>},
                  {node_3,<0.124.0>},
                  {node_4,<0.141.0>},
                  {node_5,<0.158.0>},
                  {node_6,<0.175.0>},
                  {node_7,<0.192.0>},
                  {node_8,<0.209.0>},
                  {node_9,<0.226.0>},
                  {node_10,<0.243.0>}]

2025-05-21 08:56:39.605 [IMPORTANT]
RANDOM NODE: [{node_6,<0.175.0>}]

ok
2> |

```

Figure 6: Esecuzione e output della funzione `prepare_network`, che permette di inizializzare la rete con un certo numero di nodi. Qui è mostrato l'output della creazione di 2 nodi bootstrap e 10 nodi semplici.

## 5.2 Operazioni di protocollo

Esecuzione di ping, store e find\_value

```
Eshell V13.2.2.5 (abort with ^G)
1> test:prepare_network(2,10).
Log system initialized with file logging.
2025-05-21 08:56:38.974 [INFO] Starting the network with 2 BOOTSTRAP nodes ----
Log system initialized with file logging.
Log level set with: important
2025-05-21 08:56:38.979 [IMPORTANT] Kademlia network started.
2025-05-21 08:56:39.605 [IMPORTANT]
REACHABLE NODES: [{node_1,<0.90.0>},
                  {node_2,<0.107.0>},
                  {node_3,<0.124.0>},
                  {node_4,<0.141.0>},
                  {node_5,<0.158.0>},
                  {node_6,<0.175.0>},
                  {node_7,<0.192.0>},
                  {node_8,<0.209.0>},
                  {node_9,<0.226.0>},
                  {node_10,<0.243.0>}]

2025-05-21 08:56:39.605 [IMPORTANT]
RANDOM NODE: [{node_6,<0.175.0>}]

ok
2> test:ping(<0.90.0>, <0.107.0>).
alive
3>
3> test:store(<0.158.0>, hello).
{store_response, [{node_2,<<158,204>>,<0.107.0>},
                  {bootstrap_1,<<"\f">>,<0.87.0>},
                  {node_9,<<147,90>>,<0.226.0>}]}
4>
4> Key = utils:calculate_key(ciao, 2).
<<203,156>>
5> test:find_value(<0.107.0>, Key).
{find_value_response,not_found}
6>
6> Key2 = utils:calculate_key(hello, 2).
<<"ñ">>
7> test:find_value(<0.107.0>, Key2).
{find_value_response,{hello,1747897141438,
                      {<<10,28>>,<0.158.0>}},
                      0}
8> |
```

Figure 7: Dopo aver inizializzato la rete con 2 nodi bootstrap e 10 nodi semplici tramite la funzione `prepare_network`, vengono testate le operazioni PING, STORE e FIND\_VALUE, tramite le rispettive funzioni richiamabili nel modulo `test`. Tra i parametri impostati, ci sono  $k = 3$  e  $\alpha = 1$ . `node_5` salva il valore "hello" tramite l'operazione `store`. La `store_response` contiene i 3 nodi su cui è stata fatta la store. `node_2` esegue la FIND\_VALUE prima usando la chiave `Key` (hash di "ciao") e poi la chiave `Key2` (hash di "hello"). Correttamente, la prima lookup fallisce mentre la seconda ha successo.

## Esecuzione di una find\_node

```

Eshell V13.2.2.5 (abort with ^G)
1> test:prepare_network(1,10).
Log system initialized with file logging.
2025-05-21 10:59:08.506 [INFO] Starting the network with 1 BOOTSTRAP nodes ----
Log system initialized with file logging.
Log level set with: important
2025-05-21 10:59:08.511 [IMPORTANT] Kademlia network started.
2025-05-21 10:59:09.087 [IMPORTANT]
REACHABLE NODES: [{node_1,<0.89.0>},
                  {node_2,<0.106.0>},
                  {node_3,<0.123.0>},
                  {node_4,<0.140.0>},
                  {node_5,<0.157.0>},
                  {node_6,<0.174.0>},
                  {node_7,<0.191.0>},
                  {node_8,<0.208.0>},
                  {node_9,<0.225.0>},
                  {node_10,<0.242.0>}]

2025-05-21 10:59:09.087 [IMPORTANT] RANDOM NODE: {node_3,<<133,177>>,<0.123.0>}
ok
2> Id = utils:generate_node_id(2).
<<"0b">>
3> test:find_node(<0.123.0>, Id).
{find_node_response, [{node_8,<<"Ü"/utf8>>,<0.208.0>},
                      {bootstrap_1,<<"ÆH">>,<0.87.0>},
                      {node_4,<<"é\n">>,<0.140.0>}]}
4>
4> Node = {node_3,<<133,177>>,<0.123.0>}.
{node_3,<<133,177>>,<0.123.0>}
5> test:print_buckets_with_distance(Node, Id).
2025-05-21 11:00:22.105 [IMPORTANT] node_3 (<0.123.0>):

----- BUCKETS OF node_3 (<0.123.0>)-----
--- 1: [{node_6,<0.174.0>,61766,44181},
        {node_7,<0.191.0>,34244,55319},
        {node_10,<0.242.0>,38467,52112}]
--- 2: [{node_8,<0.208.0>,17965,7166},
        {bootstrap_1,<0.87.0>,17401,7722},
        {node_4,<0.140.0>,27835,12648}]
--- 3: [{node_2,<0.106.0>,9153,32274}, {node_1,<0.89.0>,12467,28000}]
--- 4: [{node_9,<0.225.0>,4195,19888}]
--- 5: [{node_5,<0.157.0>,3661,21406}]
----- END BUCKETS -----

ok
6> |

```

Figure 8: Dopo aver inizializzato la rete con  $k = 3$ , `node_3` effettua una `FIND_NODE` su un ID generato casualmente tramite `utils:generate_node_id/1`. La `find_node_response` contiene i 3 nodi più vicini (corretto poiché  $k = 3$ ). Il fatto che siano i nodi più vicini può essere verificato chiamando `test:print_buckets_with_distance/2`, che stampa lo stato della tabella di routing del nodo assieme all'informazione sulla distanza di ogni contatto da LocalID e da un determinato TargetID (rispettivamente, penultimo e ultimo valore delle tuple stampate). Come si può osservare, i 3 nodi ricevuti sono stati inseriti nel bucket con indice 2, e correttamente hanno i valori di distanza più piccoli rispetto al ID target.

### 5.3 Republish dei valori

```

2025-05-22 20:55:02.829 [IMPORTANT] RANDOM NODE: {node_4,<<155,18>>,<0.141.0>}
ok
2> test:store(<0.141.0>,ciao).
{store_response,[{bootstrap_1,<<"\n;">>,<0.87.0>},
                  {node_1,<<221,25>>,<0.90.0>},
                  {node_2,<<86,133>>,<0.107.0>},
                  {node_3,<<146,215>>,<0.124.0>},
                  {node_5,<<"èÑ">>,<0.158.0>}]}
3> node:start_simple_node(node_prova).
{ok,{node_prova,<<"k0">>,<0.177.0>}}
4> test:print_storage(<0.177.0>).
2025-05-22 20:55:40.301 [IMPORTANT] node_prova (<0.177.0>):

----- STORAGE OF node_prova (<0.177.0>)-----
----- END STORAGE -----

ok
2025-05-22 20:55:43.126 [IMPORTANT] node_1 (<0.90.0>): timeout for value REPUBLISHING
2025-05-22 20:55:43.126 [IMPORTANT] node_3 (<0.124.0>): timeout for value REPUBLISHING
2025-05-22 20:55:43.127 [IMPORTANT] node_5 (<0.158.0>): timeout for value REPUBLISHING
2025-05-22 20:55:43.127 [IMPORTANT] node_2 (<0.107.0>): timeout for value REPUBLISHING
2025-05-22 20:55:43.127 [IMPORTANT] bootstrap_1 (<0.87.0>): timeout for value REPUBLISHING
5> test:print_storage(<0.177.0>).
2025-05-22 20:55:45.879 [IMPORTANT] node_prova (<0.177.0>):

----- STORAGE OF node_prova (<0.177.0>)-----
--- ciao (value), 1748026514126 (expiry), 1747940174129 (republish), <0.141.0> (owner)
----- END STORAGE -----

ok
6>

```

Figure 9: Innanzitutto la rete viene inizializzata con 5 nodi (non visibile nello screen). Inoltre,  $k = 10$  e `republish_interval` è impostato a 30 secondi. Viene poi salvato un valore (ciao) in  $k$  nodi, ossia tutti in questo caso. Dopodiché, viene creato un nuovo nodo denominato `node_prova`. Stampando il suo storage prima che scatti il timeout di republish, si nota che è vuoto. Aspettando il tempo di republish del valore e ristampando lo storage, si nota stavolta che il valore è presente per il nuovo nodo inserito, dimostrando il funzionamento della republish.

## 5.4 Refresh dei bucket

```

4> Node100 = {node_100,<51,28>,<0.1772,0>}.
{node_100,<51,28>,<0.1772,0>}
5> test:print_buckets_with_distance(Node100).
2025-05-21 11:31:02.466 [IMPORTANT] node_100 (<0.1772,0>):

----- BUCKETS OF node_100 (<0.1772,0>)-----
1: [{node_11,<0.259,0>,38885},
{node_18,<0.242,0>,33590},
{node_15,<0.378,0>,64990},
{node_20,<0.412,0>,58533},
{node_6,<0.174,0>,56524},
{node_15,<0.327,0>,58522},
{node_9,<0.225,0>,51861},
{node_16,<0.344,0>,46583},
{node_4,<0.140,0>,48698},
{node_17,<0.361,0>,47303}]
2: [{bootstrap_1,<0.87,0>,18636}]
3: [{node_12,<0.276,0>,9419},
{node_49,<0.985,0>,10943},
{node_48,<0.752,0>,14731},
{node_42,<0.786,0>,14045}]
4: [{node_13,<0.293,0>,5549},
{node_14,<0.310,0>,4299},
{node_65,<0.1177,0>,4520},
{node_75,<0.1313,0>,5091}]
5: [{node_47,<0.871,0>,2980},
{node_72,<0.1226,0>,2263},
{node_78,<0.1398,0>,2995},
{node_88,<0.1568,0>,3672}]
6: [{node_24,<0.488,0>,1296},
{node_29,<0.565,0>,1440},
{node_98,<0.1738,0>,1882}]
7: [{node_34,<0.650,0>,1011}]
----- END BUCKETS -----

ok
6> test:refresh_node(Node100).
2025-05-21 11:31:14.158 [IMPORTANT] node_100 (<0.1772,0>) REFRESHING bucket 0
2025-05-21 11:31:14.158 [IMPORTANT] node_100 (<0.1772,0>) REFRESHING bucket 18
2025-05-21 11:31:14.158 [IMPORTANT] node_100 (<0.1772,0>) REFRESHING bucket 18
2025-05-21 11:31:14.158 [IMPORTANT] node_100 (<0.1772,0>) REFRESHING bucket 11
2025-05-21 11:31:14.158 [IMPORTANT] node_100 (<0.1772,0>) REFRESHING bucket 2
2025-05-21 11:31:14.158 [IMPORTANT] node_100 (<0.1772,0>) REFRESHING bucket 12
2025-05-21 11:31:14.158 [IMPORTANT] node_100 (<0.1772,0>) REFRESHING bucket 13
2025-05-21 11:31:14.158 [IMPORTANT] node_100 (<0.1772,0>) REFRESHING bucket 14
2025-05-21 11:31:14.158 [IMPORTANT] node_100 (<0.1772,0>) REFRESHING bucket 2
2025-05-21 11:31:14.158 [IMPORTANT] node_100 (<0.1772,0>) REFRESHING bucket 15
2025-05-21 11:31:14.158 [IMPORTANT] node_100 (<0.1772,0>) REFRESHING bucket 6
2025-05-21 11:31:14.158 [IMPORTANT] node_100 (<0.1772,0>) REFRESHING bucket 6
2025-05-21 11:31:14.158 [IMPORTANT] node_100 (<0.1772,0>) REFRESHING bucket 4
2025-05-21 11:31:14.158 [IMPORTANT] node_100 (<0.1772,0>) REFRESHING bucket 7
2025-05-21 11:31:14.158 [IMPORTANT] node_100 (<0.1772,0>) REFRESHING bucket 8
2025-05-21 11:31:14.158 [IMPORTANT] node_100 (<0.1772,0>) REFRESHING bucket 9
2025-05-21 11:31:14.169 [IMPORTANT] node_100 (<0.1772,0>) REFRESH COMPLETE
node_100 (<0.1772,0>) REFRESHED SUCCESSFULLY.
ok
8>

```

Figure 10: La rete è inizializzata con 100 nodi (non visibile negli screen). Viene testato il refresh dei bucket dell'ultimo nodo entrato `node_100`. Si stampa la tabella di routing del nodo prima (primo screen) e dopo (terzo screen) aver effettuato un refresh dei buckets (secondo screen). Come si può osservare, la tabella dopo il refresh contiene più entrate di quella prima, provando che la refresh ha avuto un effetto positivo sul popolamento della tabella.

## 6 Risultati sperimentali

In questa sezione si descrivono i risultati raggiunti tramite l'esecuzione di esperimenti sull'implementazione. Sono state svolte simulazioni per misurare il tempo medio di lookup, il tempo medio di lookup in presenza di guasti (fino a  $k - 1$  nodi) e il tempo di join medio di un nuovo nodo nella rete.

### 6.1 Tempo di lookup medio

È stato misurato il tempo di lookup medio, espresso in millisecondi e in numero di hop, per l'operazione di ricerca di un valore. La simulazione è stata ripetuta incrementando esponenzialmente il numero di nodi  $N$  nella rete, fino ad arrivare ad oltre 4000 nodi. Per ogni valore di  $N$  si è presa la media di 100 misurazioni svolte. Come nodo di test è stato selezionato un nodo casuale e, per ogni misurazione, è stato salvato un valore casuale nella rete tramite l'operazione di store. I parametri della simulazione sono riportati in tabella 1.

In figura 11 è riportato il grafico del tempo di lookup al variare del numero dei nodi. Si può osservare come il tempo di risposta cresca all'incirca linearmente rispetto al logaritmo del numero di nodi  $N$ , confermando la complessità teorica  $O(\log_2 N)$  di Kademlia.



Parametro	Valore
Bootstrap nodes	5
Misurazioni per prova	100
Lunghezza ID	16 bit
$K$	10
$\alpha$	3
Refresh	No
Nodi	[8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096]

Table 1: Parametri sperimentali per la prima simulazione.

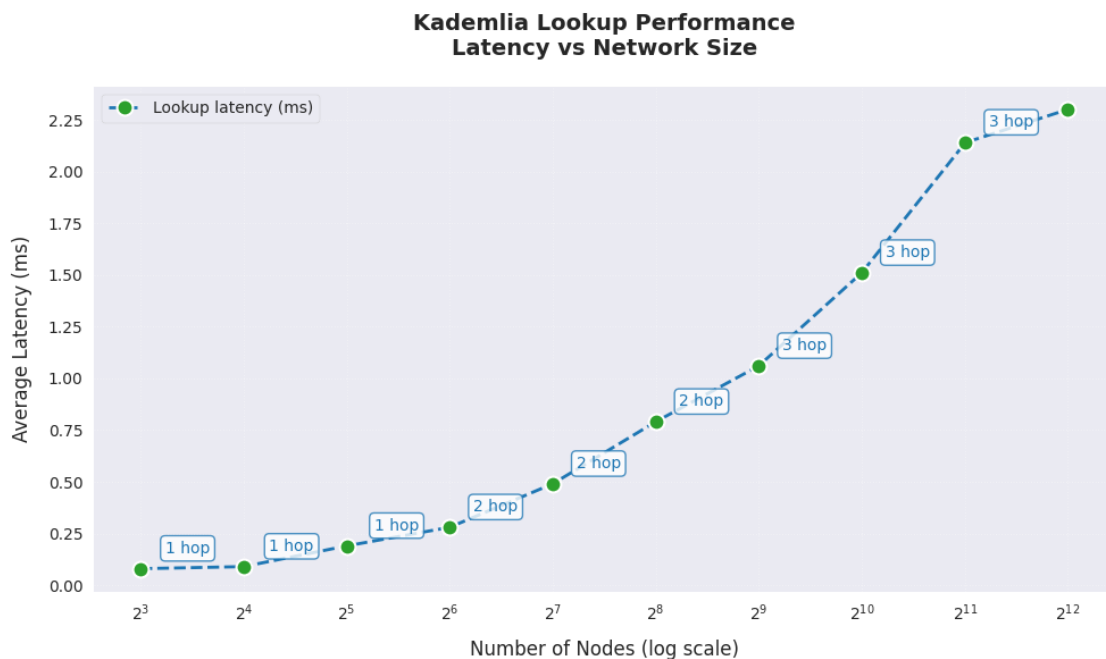


Figure 11: Tempo di lookup medio in millisecondi in funzione di  $\log_2 N$ , dove  $N$  è il numero di nodi totali. Viene riportato per ogni punto anche il numero di hop medio. Si può osservare che la curva è quasi lineare, pertanto la complessità è circa logaritmica.

## 6.2 Tempo di join medio

Un altro esperimento svolto ha permesso di misurare il tempo di join medio di un nodo nella rete in funzione del numero di nodi totali. Il tempo di join dipende dal tempo di lookup, poiché inizialmente il nodo deve contattare i propri vicini, quindi ci si aspettano dei risultati simili in termini di complessità. La simulazione è stata svolta attivando il refresh dei bucket. In questo modo, ciascun nodo entrante, oltre a fare una lookup di se stesso, effettua una lookup su un ID casuale per ciascuno dei suoi bucket, andando così a scoprire anche i nodi più lontani. I parametri della simulazione sono riportati in tabella 2.

In figura 12 sono visualizzati i risultati, riportando l'andamento del tempo di join in funzione del logaritmo del numero di nodi totali. Anche in questo caso, si ha un andamento all'incirca lineare, suggerendo che anche il tempo di join ha complessità logaritmica.

Parametro	Valore
Bootstrap nodes	5
Misurazioni per prova	20
Lunghezza ID	16 bit
$K$	20
$\alpha$	3
Refresh	Si
Nodi	[8, 16, 32, 64, 128, 256, 512, 1024, 2048]

Table 2: Parametri sperimentali per la seconda simulazione.

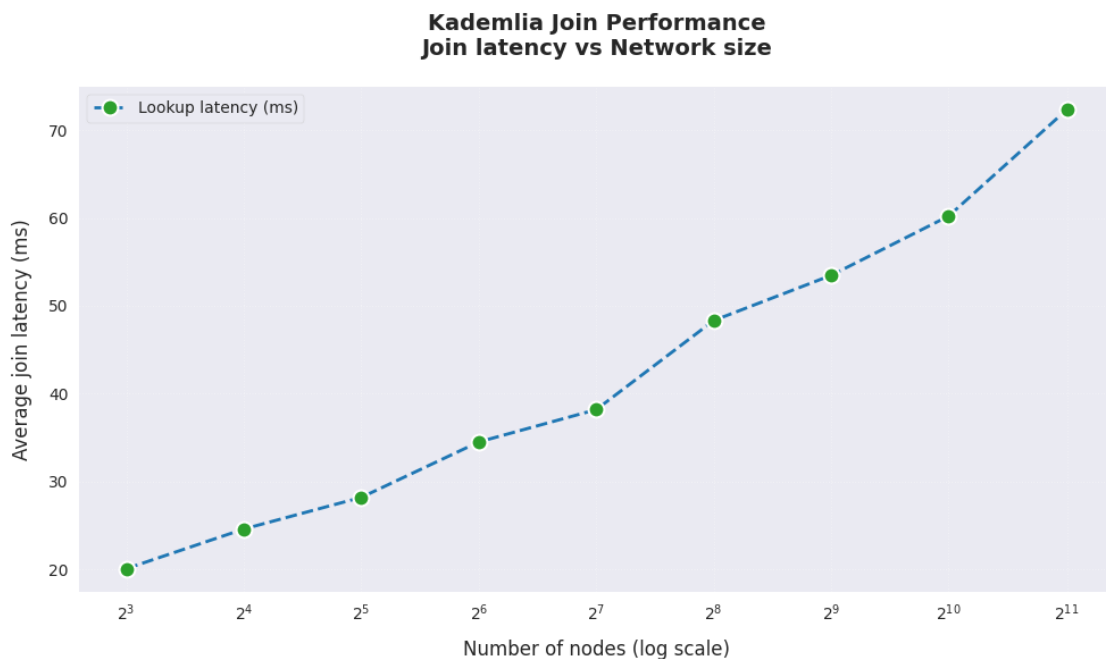


Figure 12: Tempo di join medio (su 20 misurazioni) in millisecondi, in funzione di  $\log_2 N$ , dove  $N$  è il numero di nodi totali. Si può osservare che la curva è quasi lineare, pertanto la complessità è logaritmica.

### 6.3 Lookup in presenza di nodi guasti

Si è voluto misurare il tempo di lookup medio in presenza di guasti nei nodi, per osservare quanto la rete sia resiliente agli errori. La rete simulata contiene 1000 nodi. Si fanno fallire solo fino a  $k - 1$  nodi più vicini alla chiave, diversi dai nodi di bootstrap. Si misura poi il tempo di lookup medio facendo variare il numero di nodi falliti. Per ogni numero di nodi falliti, si salva un valore casuale nella rete tramite l'operazione di store e si effettuano 100 misurazioni di lookup per quel valore. Ciascuna misurazione viene svolta da un nodo casuale tra quelli rimasti in vita. I parametri sono quelli in tabella 3.

La tabella 4 riporta i risultati ottenuti. Le misurazioni sul lookup medio non hanno portato a risultati significativi. L'ipotesi che si voleva dimostrare era quella secondo cui il tempo di lookup medio in millisecondi aumentasse in maniera proporzionale all'aumentare del numero di nodi falliti. Questa ipotesi non è stata confermata dagli esperimenti, che invece suggeriscono un comportamento della rete più caotico in presenza di guasti. Si è osservato però un aumento graduale del tasso di fallimento delle value lookup. Questo

Parametro	Valore
Bootstrap nodes	5
Misurazioni per prova	100
Lunghezza ID	16 bit
$K$	5
$\alpha$	1
Refresh	Si
Nodi	1000

Table 3: Parametri sperimentali per la terza simulazione.

fenomeno è visualizzato nel grafico in figura 13, dove si osserva che l'andamento del numero di tentativi totali necessari a raggiungere almeno 100 lookup di successo (valore trovato) aumenta in funzione del numero di nodi falliti, da 1 fino a  $k - 1$ .

Nodi falliti	Tempo medio (ms)	Hop medi	Tentativi totali
0	0.26	1	100
1	0.04	2	280
2	0.05	1	332
3	0.05	2	407
4	0.05	2	600

Table 4: Risultati medi del lookup con guasti ai nodi

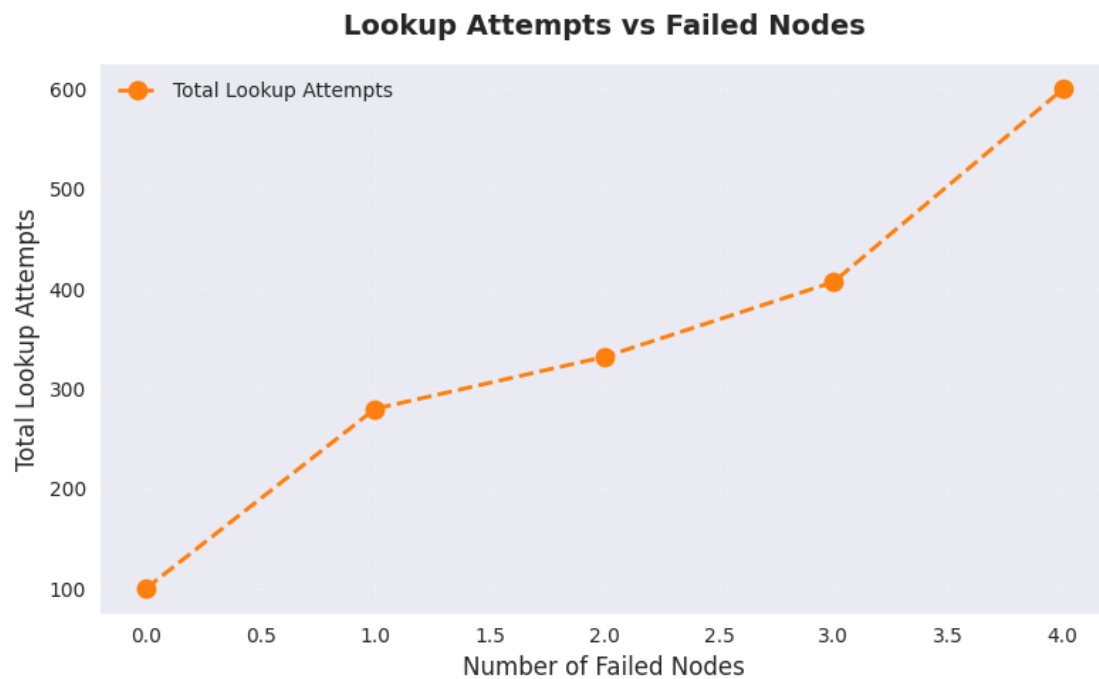


Figure 13: Numero di tentativi necessari a svolgere 100 value lookup con successo, in funzione del numero di nodi falliti nella rete. Si osserva un aumento in correlazione al numero di nodi falliti da 1 a  $k - 1$ .

## 7 Conclusioni

Questo lavoro ha presentato un'implementazione pratica delle funzionalità fondamentali di Kademlia. Il linguaggio Erlang, con il suo modello ad attori, si è rivelato particolarmente adatto ad un progetto di questo tipo, in quanto permette di modellare agevolmente e in modo efficiente la natura asincrona e concorrente di Kademlia.

I test di validazione hanno confermato la correttezza delle principali operazioni di protocollo. Inoltre, gli esperimenti condotti hanno verificato empiricamente la complessità logaritmica delle operazioni, in linea con le aspettative teoriche di Kademlia.

Tuttavia, i test in scenari con nodi guasti hanno rivelato delle limitazioni nella resilienza del sistema. Sebbene la rete mantenga un certo grado di funzionalità, il crescente tasso di fallimento suggerisce che l'implementazione potrebbe essere migliorata per gestire in modo più robusto situazioni di fault. Per aumentare la resilienza potrebbe essere utile introdurre meccanismi di caching e aumentare il numero di nodi di replicazione.

## References

- [1] Petar Maymounkov and David Mazières. Kademlia: A peer-to-peer information system based on the xor metric. In *International Workshop on Peer-to-Peer Systems*, pages 53–65. Springer, 2002.