

Koorde DHT

Progetto B4: Small-scale DHT System

Flavio Simonelli 0365333

Grafo di de Bruijn

L'idea alla base di Koorde

Cos'è un Grafo di De Bruijn

Un grafo di de Bruijn di base k e dimensione n è un grafo diretto in cui:

- ▶ Ogni **nodo** rappresenta una stringa di n simboli su un alfabeto di k simboli
- ▶ Ogni **arco** è orientato e collega due nodi se la stringa di destinazione può essere ottenuta tramite la seguente formula:

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

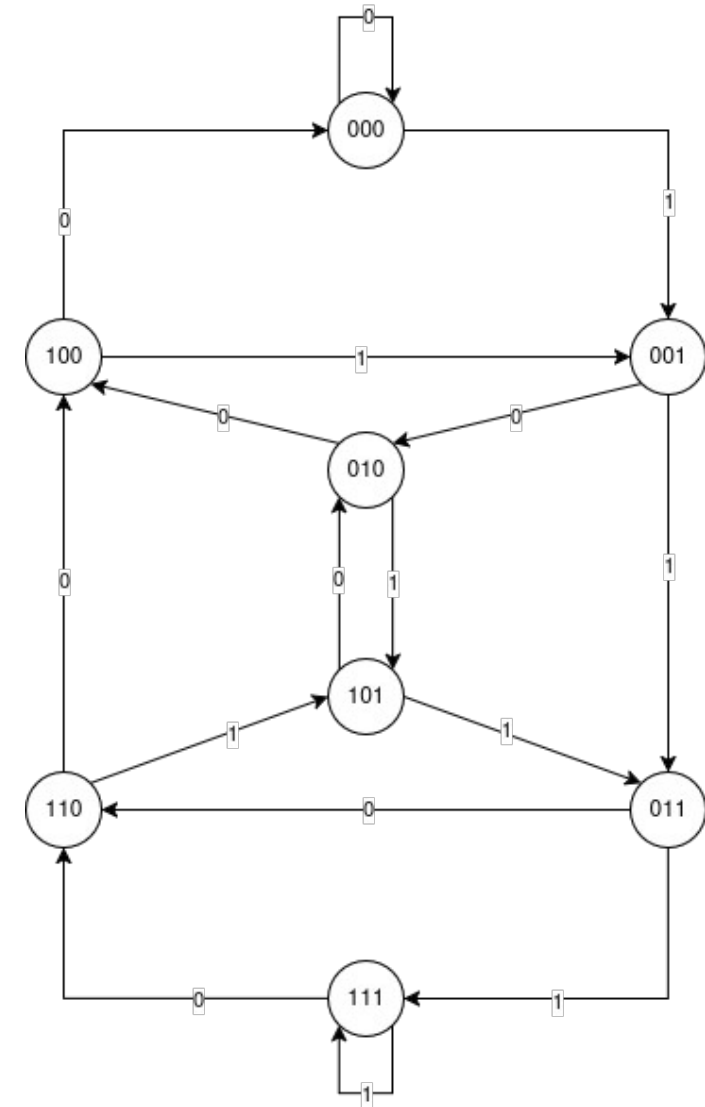
Esempio con $k = 2$ e $n = 3$

$$m \rightarrow (2m) \bmod 2^n$$

$$m \rightarrow (2m+1) \bmod 2^n$$

Proprietà

- ▶ Numero di nodi: k^n
- ▶ Diametro: $\log_k(k^n) = n \rightarrow$ solo n passi per raggiungere qualunque nodo
- ▶ Ogni nodo ha k archi uscenti e k archi entranti

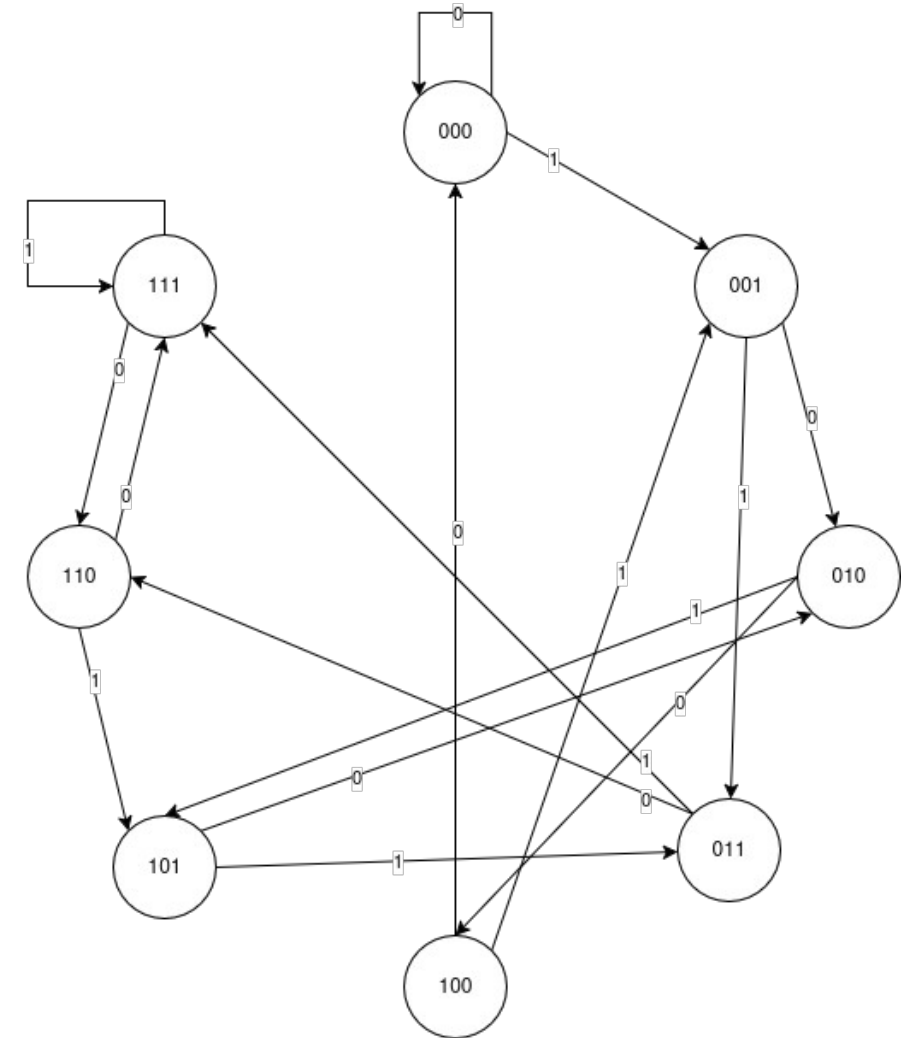


Riorganizzazione ad anello

- Ogni arco può essere ottenuto **shiftando a sinistra** di una posizione la stringa di origine e **aggiungendo** un nuovo simbolo alla fine

$$(t_1, t_2, \dots, t_n) \rightarrow (t_2, \dots, t_n, 0),$$

$$(t_1, t_2, \dots, t_n) \rightarrow (t_2, \dots, t_n, 1).$$



Differenza con Chord

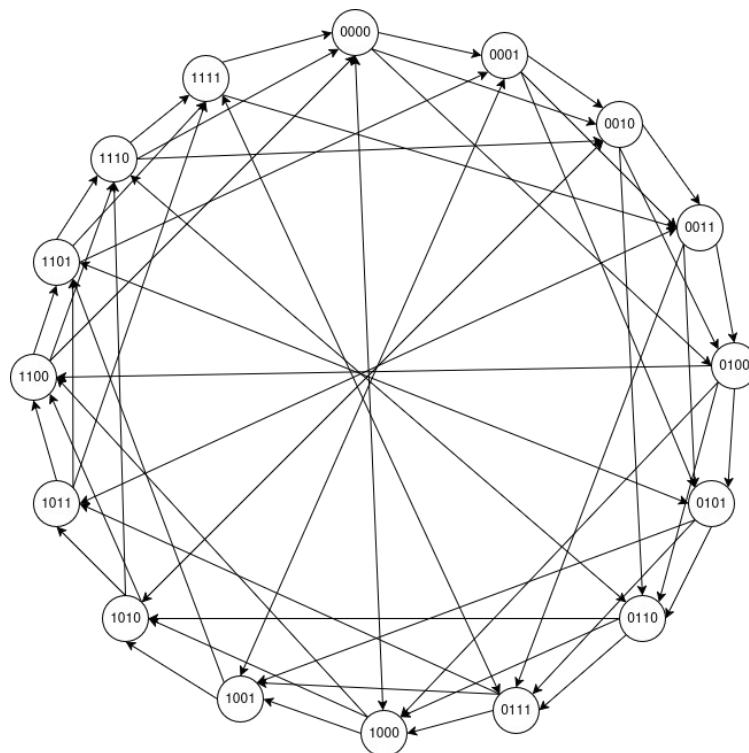
Grado di un nodo: è il numero di **connessioni dirette** (link di routing) che il nodo mantiene con altri nodi della rete (la conoscenza dei suoi vicini)

$$\text{degree}(v) = \{u \in V : v \rightarrow u\}$$

Finger Table nodo
0000

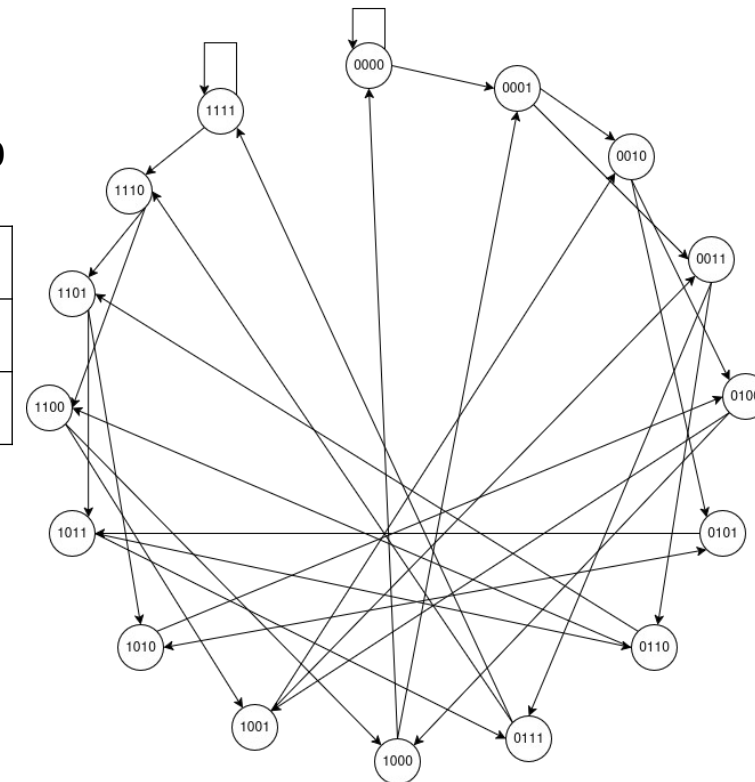
i	Nodo
1	0001
2	0010
3	0100
4	1000

$$\text{finger}_{i,m} = (m + 2^{(i-1)}) \bmod 2^n$$



Routing Table nodo
0000

i	Nodo
0	0000
1	0001



Routing table con grado k

Nel caso più generale con un grafo di de brujin di grado k *abbiamo*:

Tabella di routing del nodo m

i	Nodo
0	$(km) \bmod(k^n)$
1	$(km + 1) \bmod(k^n)$
2	$(km + 2) \bmod(k^n)$
...	...
$k-1$	$(km + k-1) \bmod(k^n)$

Chord

$O(\log n)$

Logaritmico in base al numero di
nodi nella rete

Koorde

$O(k)$

Costante in base al grado del
grafo di de Brujin utilizzato

Koorde su spazio sparso

- ▶ In una DHT solo una frazione dei nodi 2^b è reale
- ▶ Vengono introdotti I ***nodì immaginari***
- ▶ Ogni nodo reale **m** è responsabile di tutti I nodi immaginari che sono in **(m,succ(m)]**
- ▶ Spesso ad un nodo reale corrispondono diversi nodi immaginari

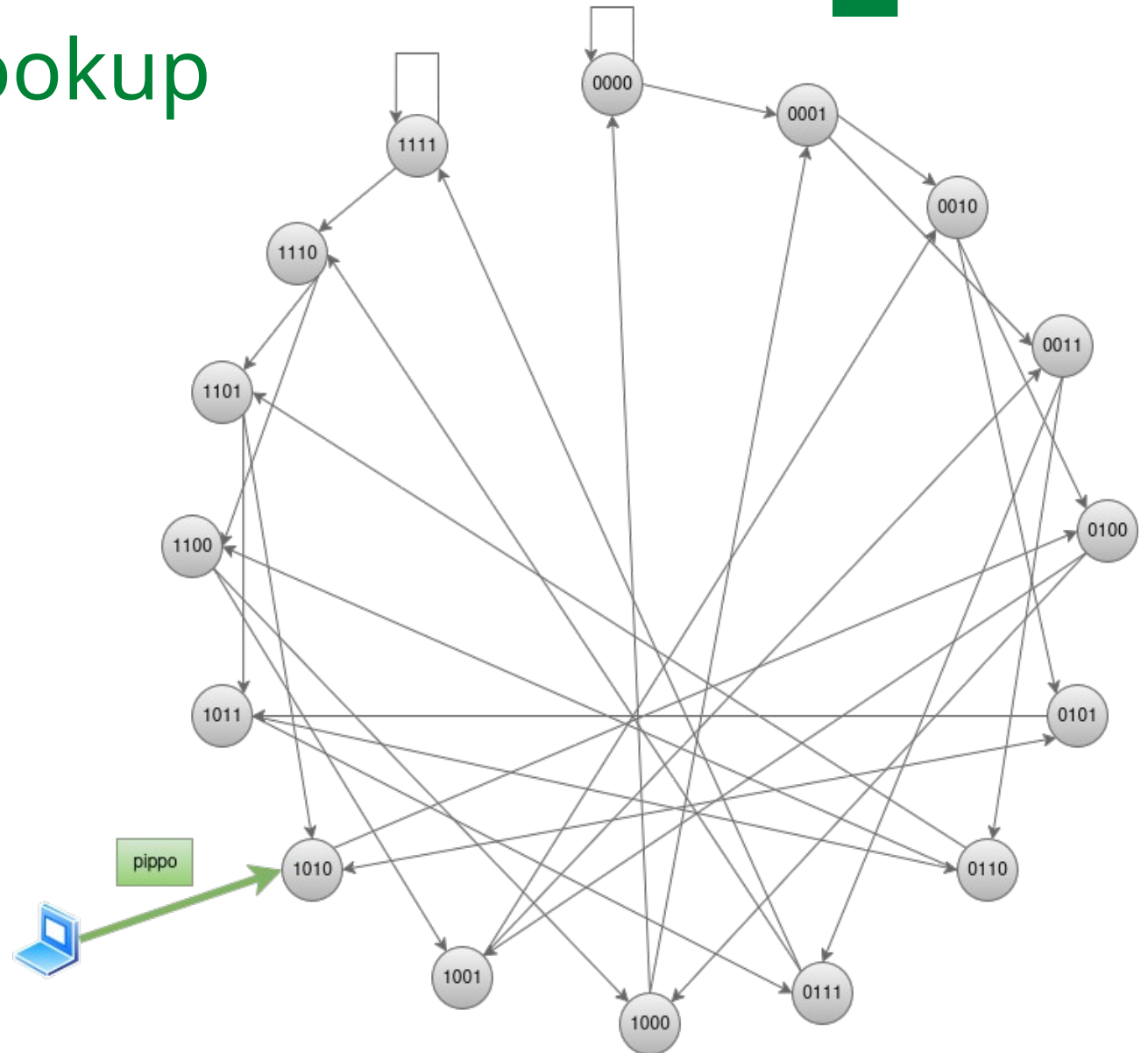
Tabella di routing del nodo m

i	Nodo
0	$\text{pred}\{(km) \bmod(k^n)\}$
1	$\text{pred}\{(km + 1) \bmod(k^n)\}$
2	$\text{pred}\{(km + 2) \bmod(k^n)\}$
...	...
k-1	$\text{pred}\{(km + k-1) \bmod(k^n)\}$

Algoritmo di Routing

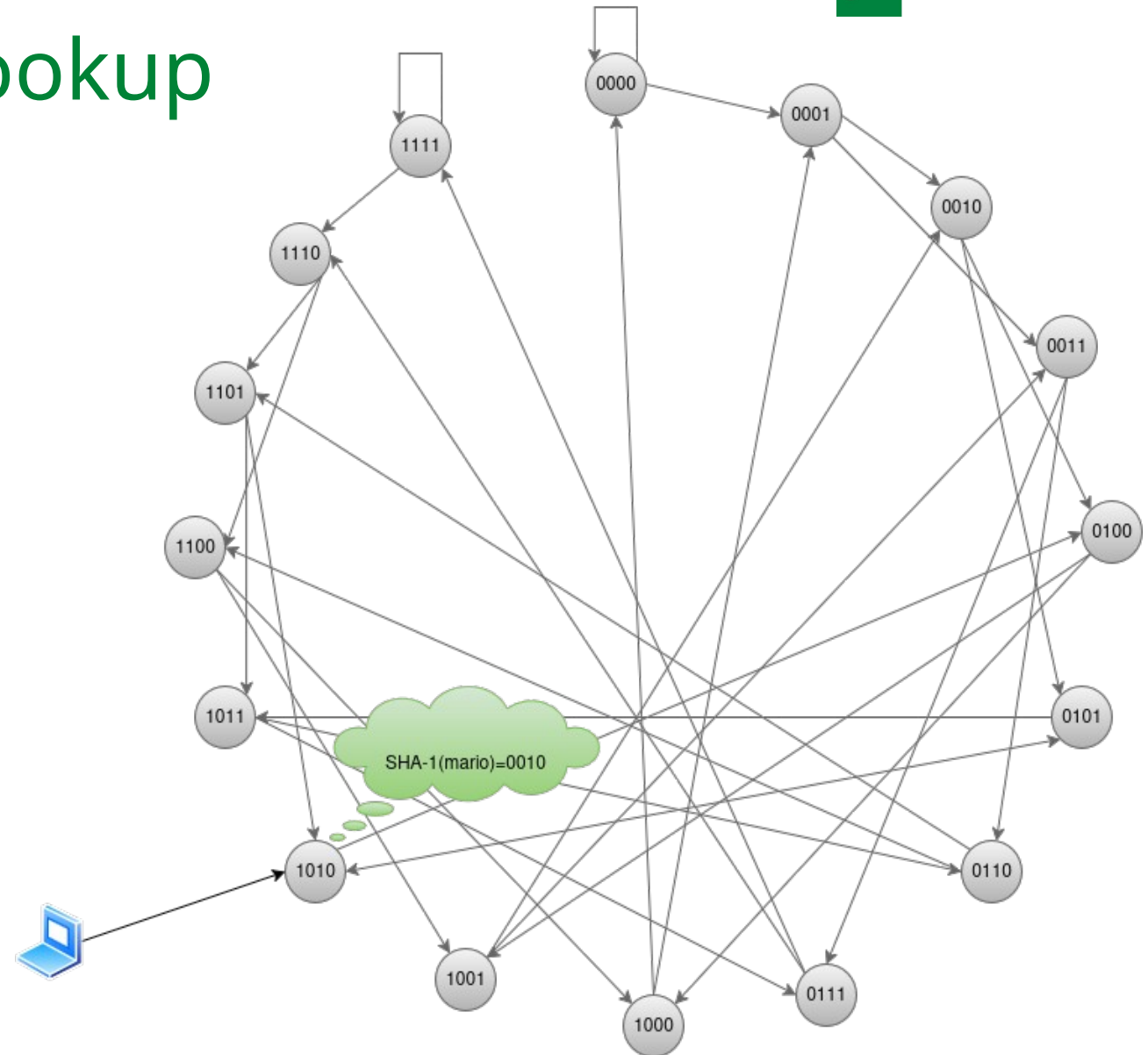
Idea dell'algoritmo di Lookup

- Il client conosce l'indirizzo del nodo con identificativo "1010"
- Lo contatta chiedendo di trovare la risorsa **pippo**



Idea dell'algoritmo di Lookup


- Il nodo “server” traduce la chiave tramite l'algoritmo di hashing **sha-1**, troncando il risultato di 128 ai primi **4 bit** più significativi




Idea dell'algoritmo di Lookup

- Prende la cifra più significativa della chiave target e il resto shiftato di un bit:

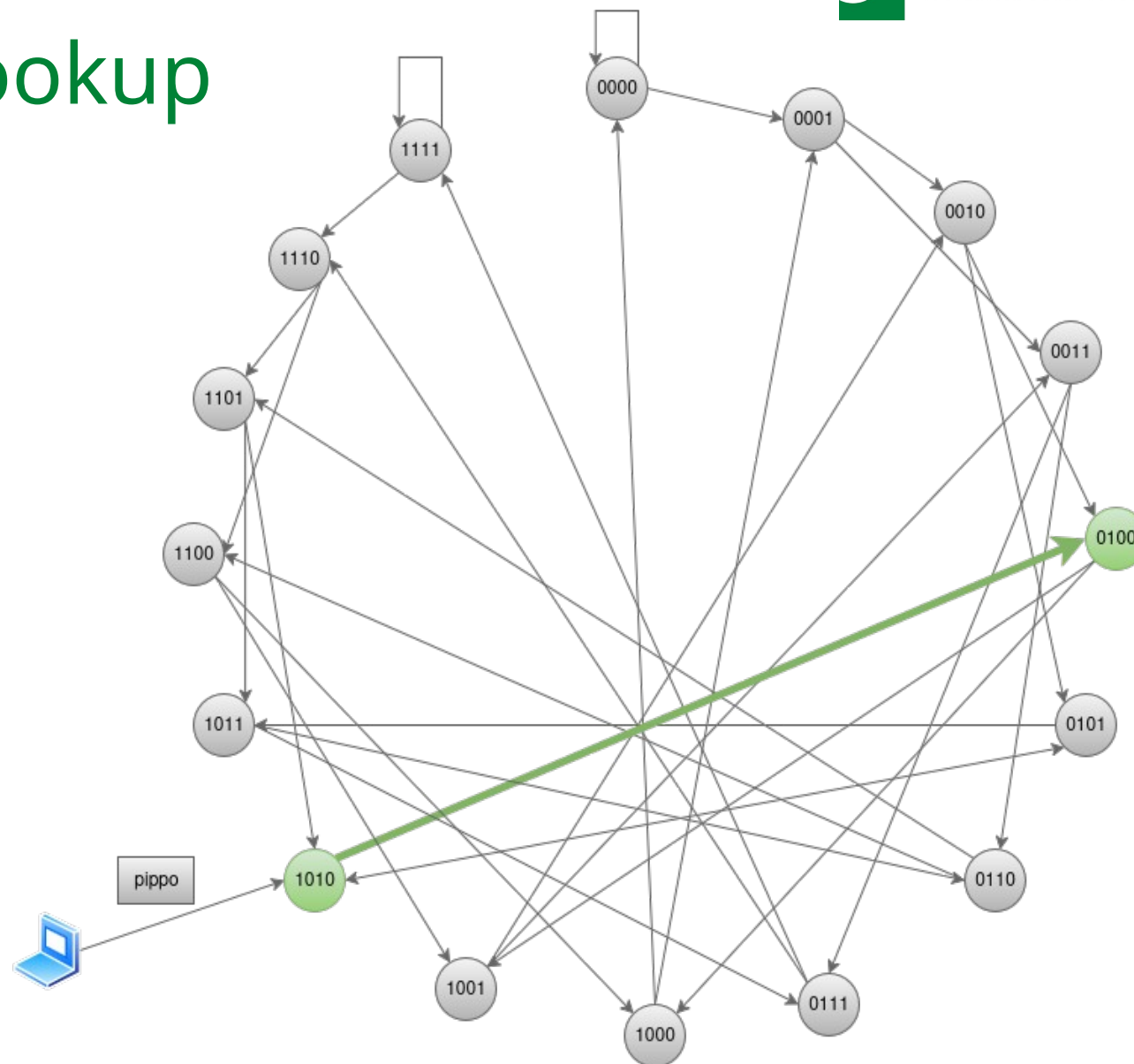
$K = 0010$

 0 = digit

 0100 = kshift



- Shifta il suo id di 1 bit e somma il valore del digit appena estratto:

$1010 \rightarrow 0100$




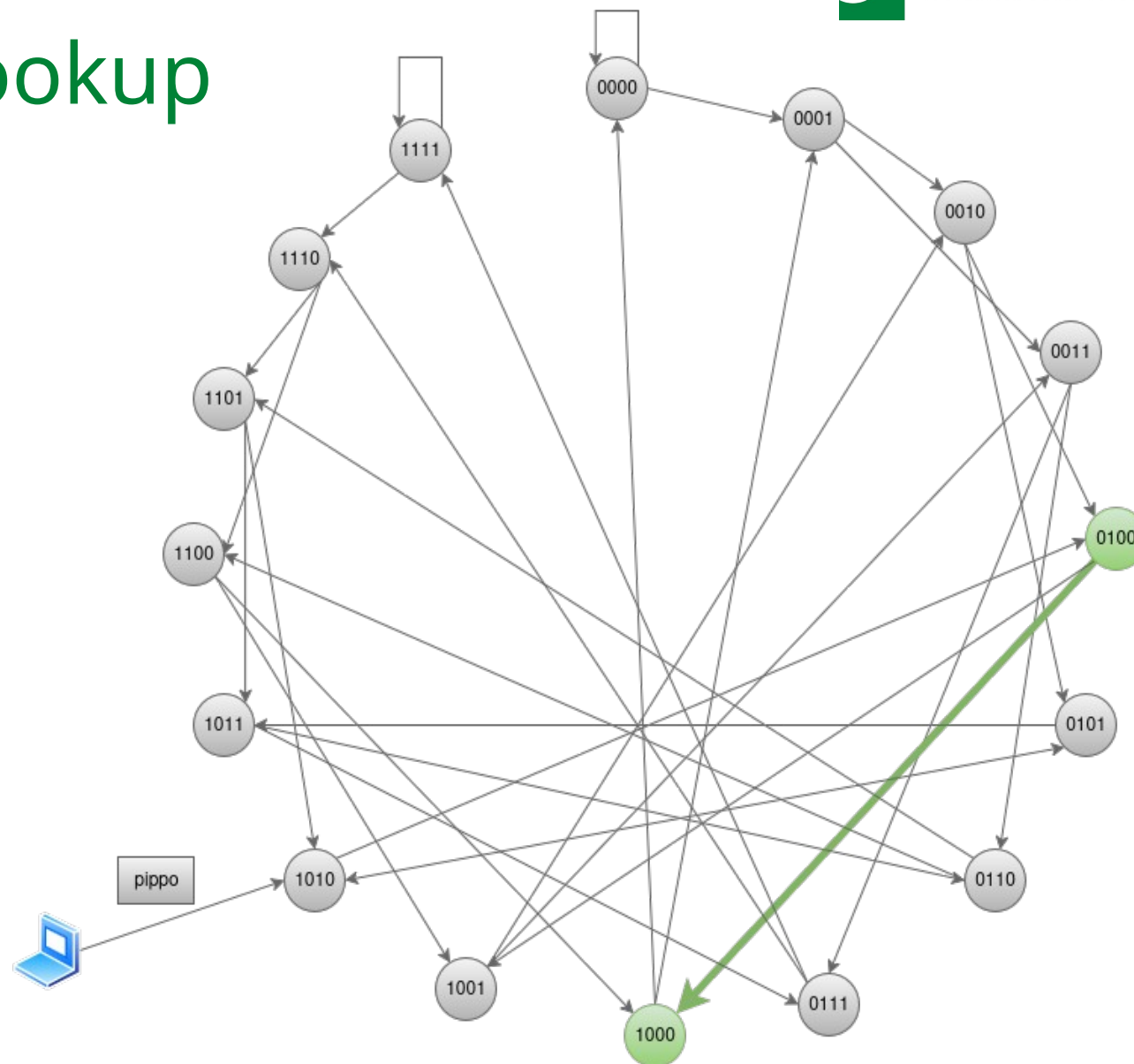
Idea dell'algoritmo di Lookup

- Prende la cifra più significativa del kshift ricevuto e il resto shiftato di un bit:

Kshift = 0100  0 = digit
 1000 = newkshift



- Shifta il suo id di 1 bit e somma il valore del digit appena estratto:

0100  1000



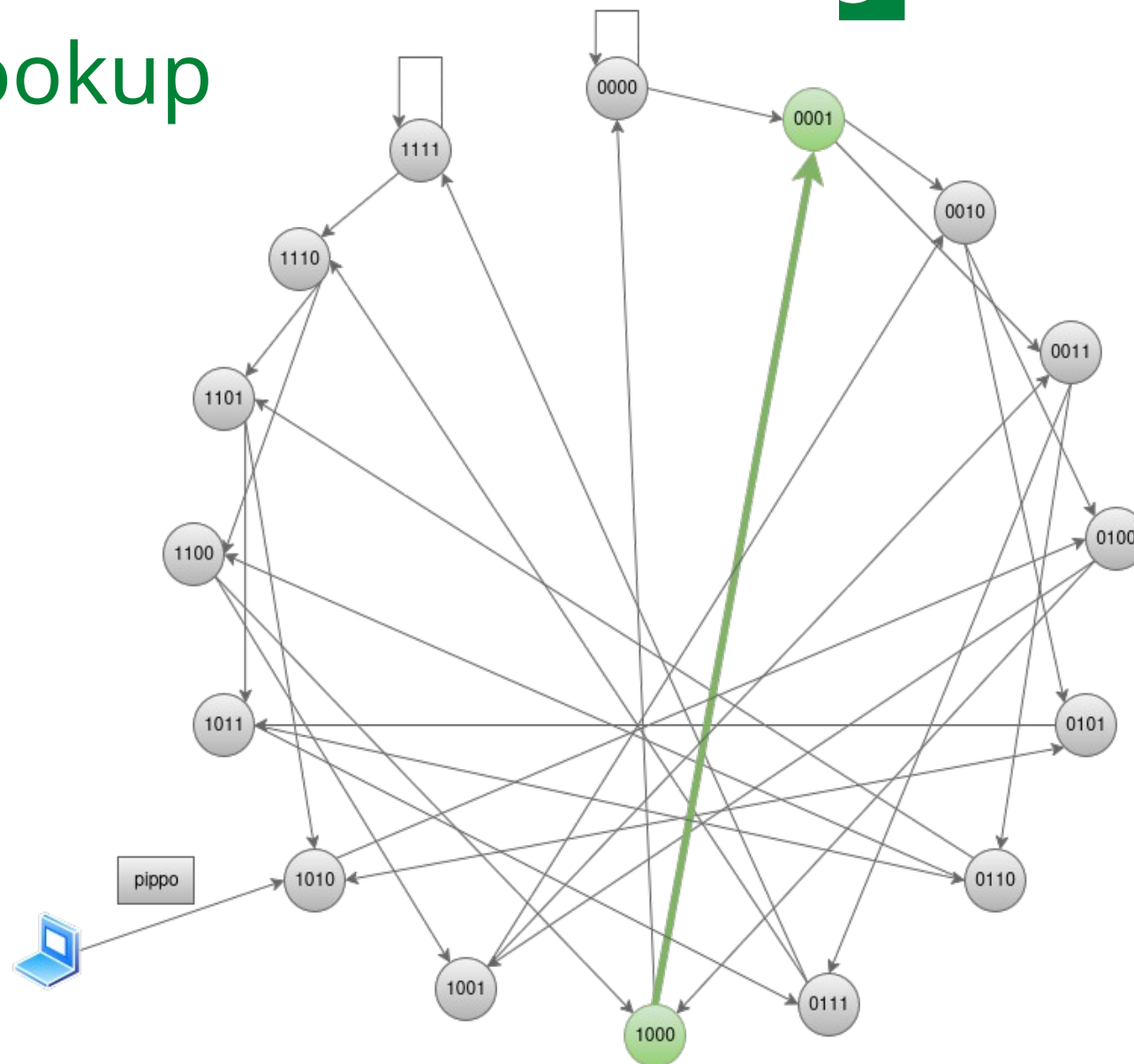
Idea dell'algoritmo di Lookup

- Prende la cifra più significativa del kshift ricevuto e il resto shiftato di un bit:

Kshift = 1000  1 = digit
 0000 = newkshift



- Shifta il suo id di 1 bit e somma il valore del digit appena estratto:

1000  0001



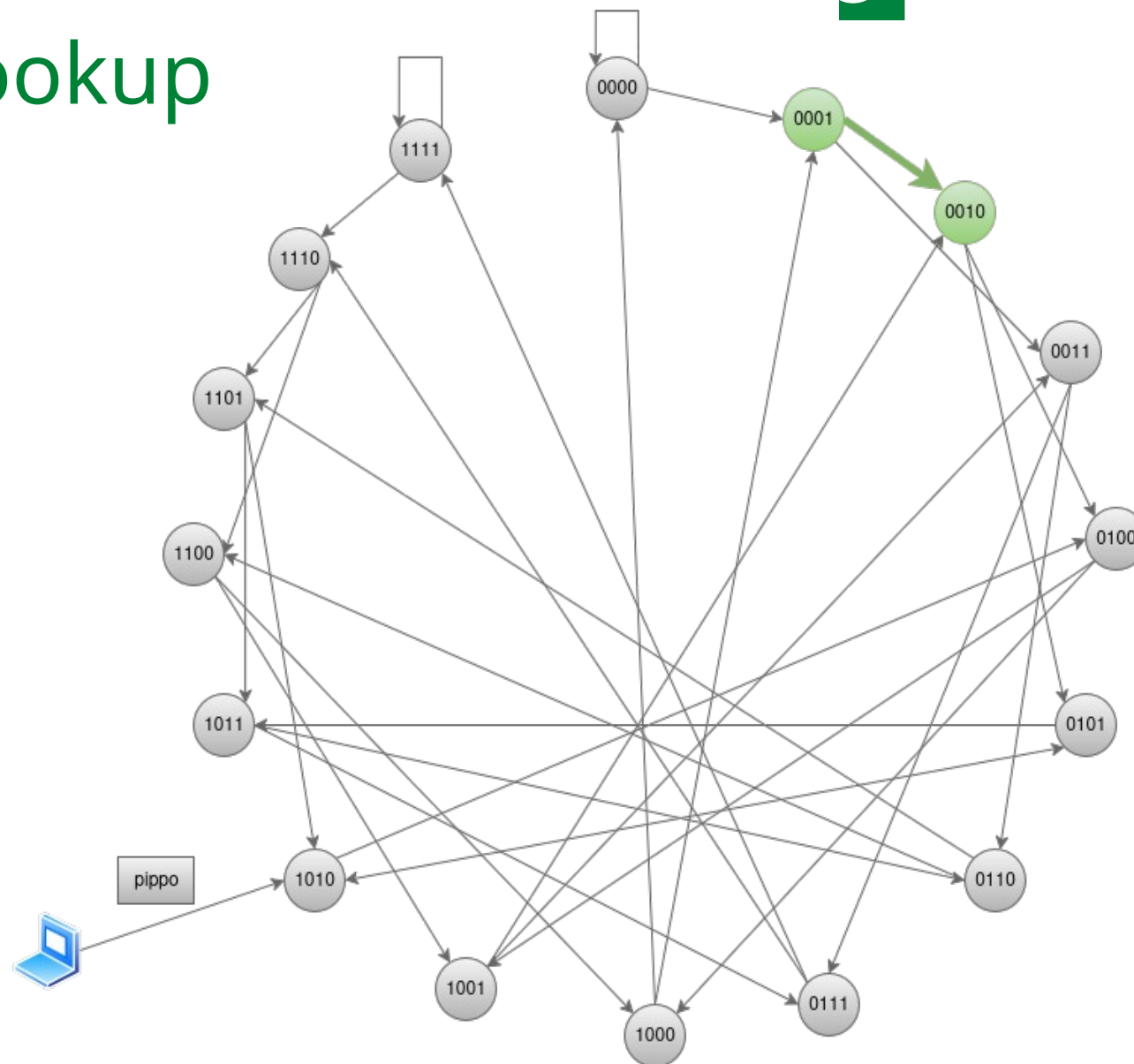
Idea dell'algoritmo di Lookup

- Prende la cifra più significativa del kshift ricevuto e il resto shiftato di un bit:

Kshift = 0000  0 = digit
 0000 = newkshift

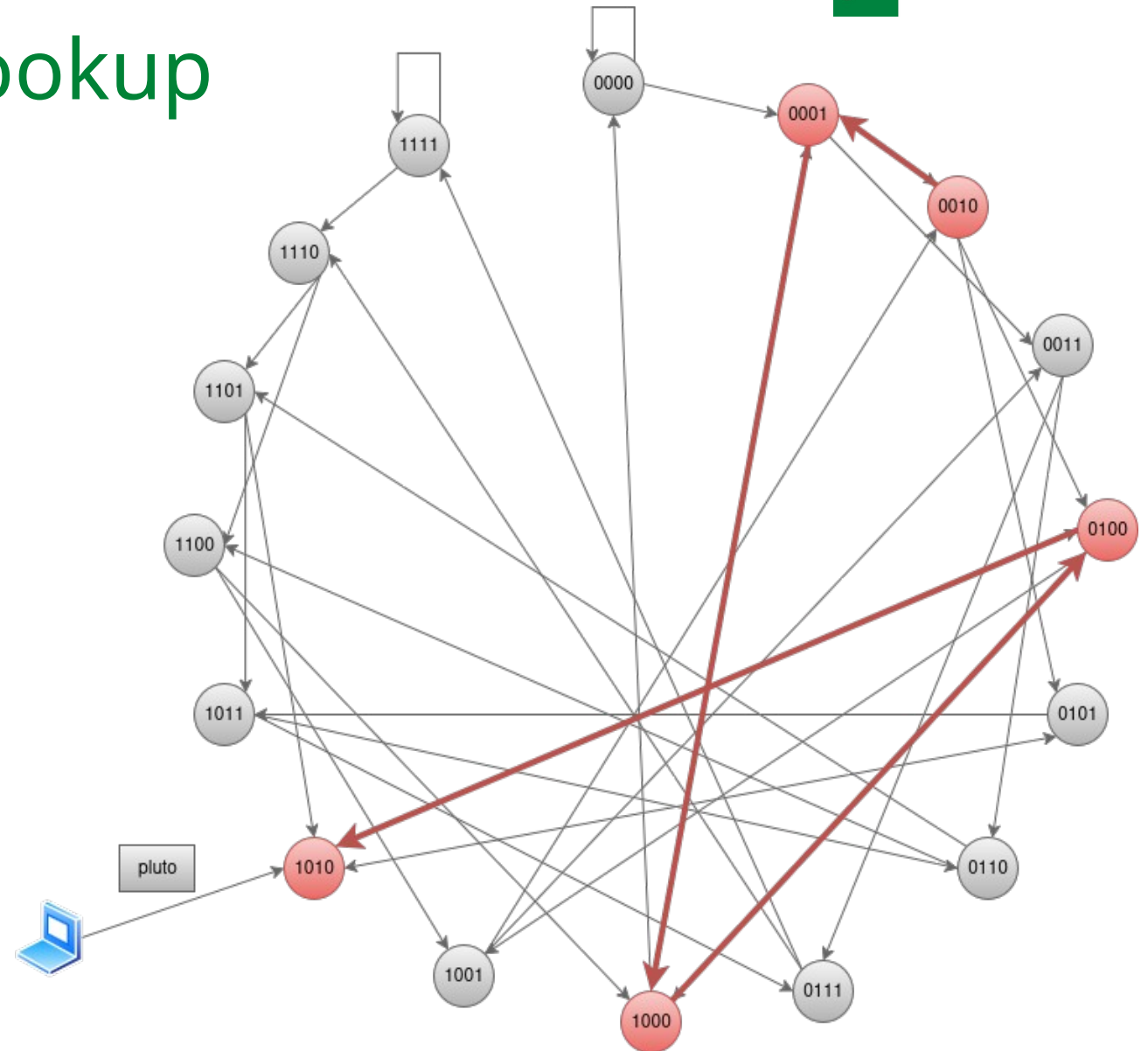
- Shifta il suo id di 1 bit e somma il valore del digit appena estratto:

0001  0010



Idea dell'algoritmo di Lookup

- Il nodo “0010” è **esattamente** il successore della chiave “0010”



Algoritmo di routing $k=2$

Considerando come:

- **k** : chiave target
 - **$Kshift$** : chiave k shiftata (resto rimanente dalle precedenti iterazioni)
 - **l** : nodo immaginario richiesto
 - **Successor**: successore immediato di m
 - \circ : concatenazione
- **Efficienza:** $O(\log_2 n)$

```
procedure  $m.LOOKUP(k, kshift, i)$   
  if  $k \in (m, successor]$  then return ( $successor$ )  
  else if  $i \in (m, successor]$  then return (  
     $d.lookup(k,$   
       $kshift \ll 1,$   
       $i \circ topBit(kshift)))$   
  else return ( $successor.lookup(k, kshift, i)$ )
```

Algoritmo di routing generale

Considerando come:

- **k** : chiave target
- **$Kshift$** : chiave k shiftata (resto rimanente dalle precedenti iterazioni)
- **l** : nodo immaginario richiesto
- **Successor**: successore immediato di m
- \circ : concatenazione
- **b** : grado del grafo
- **Efficienza**: $O(\log_b n)$

```
procedure  $m$ .LOOKUP( $k, k_{shift}, i$ )  
  if  $k \in (m, successor]$  then  
    return ( $successor$ )  
  else if  $i \in (m, successor]$  then  
     $nextDigit \leftarrow topDigit_b(k_{shift})$   
     $new\_i \leftarrow i \circ nextDigit$   
    return  $d$ .LOOKUP( $k, k_{shift} \ll_b 1, new\_i$ )  
  else  
    return ( $successor$ .LOOKUP( $k, k_{shift}, i$ ))  
  end if  
end procedure
```

Prestazioni

Osservazione

In un sistema con n nodi e grado massimo d il numero di salti richiesti è **almeno** $H_{min} = \log_d n - 1$

Dimostrazione

Deriva dalla serie geometrica: $N(h) = 1 + d + d^2 + \dots + d^h = \frac{d^{h+1} - 1}{d - 1}$

Dove supponiamo che non ci sono sovrapposizioni (ogni nodo raggiunge nuovi nodi)

$$\frac{d^{h+1} - 1}{d - 1} \geq n \rightarrow d^{h+1} \geq n \rightarrow h \geq \log_d n - 1$$

Prestazioni

Possiamo approssimare a meno di costanti: $H_{min} = \log_d n - 1 = O(\log_d n) = O\left(\frac{\log n}{\log d}\right)$

Prestazioni di Koorde:

- Se $d=2$: $H_{min} = \log_2 n = O(\log n)$
- Se $d = O(\log n)$: $H_{min} = \log_{O(\log n)} n = \frac{\log n}{\log(\log n)} = O\left(\frac{\log n}{\log \log n}\right)$

Implementazione

Linguaggio GO

Identificativi

- Viene utilizzato: `type ID []byte`
- Permette di essere molto più efficiente per le operazioni di shift e somma modulare rispetto ad altri tipi di dati
- Permette di modificare tramite parametro la lunghezza degli identificativi usati nella DHT
- Richiede la completa implementazione manuale delle operazioni bit a bit
- Richiede la corretta gestione dei bit extra quando gli id sono composti da un numero di bit non multiplo del byte
- Le operazioni fondamentali:

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

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

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

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

Nodo

- Ogni nodo ha come componente principale un'istanza della struttura Node

```
type Node struct {  
    lgr logger.Logger  
    rt  *routingtable.RoutingTable  
    s   *storage.Storage  
    cp  *client2.Pool  
}
```

Composta da:

- Un **logger**: per i log di sistema
- Una **routing table**: contiene le informazioni necessarie per il routing e stabilizzazione
- Uno **storage** in-memory: semplice storage per dare un'utilità concreta alla DHT
- Un **clientPool**: gestire il corretto riutilizzo delle connessioni gRPC

Routing Table

- La routing table è la struttura che contiene tutte le informazioni per il routing delle richieste e per i protocolli di stabilizzazione dei nodi

Abbiamo all'interno:

- Space**: definisce il contratto della DHT
- Self**: definisce id e address del possessore della routing table
- Successor List**: lista degli immediati successori di self
- Predecessor**: puntatore al predecessore di self
- DeBruijn**: lista dei link ai nodi del grafo di de Bruijn

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

Concorrenza a grana fine con slice fissa e RWMutex per entry

Client Pool

- ▶ Il client pool è una struttura che mantiene I riferimenti alle connessioni gRPC già aperte verso altri nodi
- ▶ Il suo scopo è quello di riutilizzare le connessioni esistenti per risparmiare risorse
- ▶ La concorrenza è gestita con un singolo Mutex poichè la zona critica è molto ridotta

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

Storage

- ▶ Semplice storage in memory come dimostrazione di possibile utilizzo della rete
- ▶ Implementato come una mappa con chiave basata sulla rappresentazione esadecimale di id

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

```
type Resource struct {  
    Key      ID  
    RawKey   string  
    Value    string  
}
```

Worker periodici

- Ogni nodo esegue in background tre loop periodici indipendenti, responsabili della stabilità topologica e della coerenza dei dati nel tempo
- Tutti i worker sono governati da un context di shutdown (graceful stop) e da intervalli configurabili

Worker	Scopo	Tipologia	Intervallo	Funzioni
Chord Stabilizer	Successor/ Predecessor upkeep	Topologia	chordInterval	StabilizeSuccessor FixSuccessorList checkPredecessor
De Bruijn Fixer	Aggiorna collegamenti de Bruijn	Routing	DeBruijnInterval	fixDebruijn
Resource Repair	Ribilancia ownership dei dati	Storage	storageInterval	resourceRepair

Servizi gRPC

- Servizio: **DHT**
 - **FindSuccessor**
 - **GetPredecessor**
 - **GetSuccessorList**
 - **Notify**
 - **Ping**
 - **Store**
 - **Retreive**
 - **Remove**
 - **Leave**
- Servizio: **Client**
 - **Put**
 - **Get**
 - **Delete**
 - **GetStore**
 - **GetRoutingTable**
 - **Lookup**

Join

- Consentire a un nuovo nodo di entrare in una rete KoordeDHT esistente, inizializzando correttamente la propria tabella di routing e i puntatori De Bruijn
- Simile al protocollo Chord
- I nodi di bootstrap possono essere statici, inseriti in fase di configurazione o tramite DNS **Route53**

Procedura:

1) Bootstrap

- Tenta connessione sequenziale ai peer forniti
- FindSuccessorStart(self.ID) fino a trovare il successore valido

2) Handshake con il successore

- Richiede GetPredecessor() al successore
- Esegue Notify() verso il successore

3) Aggiornamento routing table

- Imposta Predecessore e Successore nella routing table
- Aggiorna la successor list (fixSuccessorList())
- Inizializza I de Bruijn pointer (fixdeBruijn())

Leave

- Permettere l'uscita controllata di un nodo, preservando consistenza topologica e integrità dei dati
- Garantisce che ogni risorsa rimanga nella DHT anche dopo l'uscita

Procedura:

- 1) Verifica configurazione
 - Se nodo singolo → No-Op
- 2) Notifica al successore
 - Notifica al successore la sua uscita (Leave(self))
- 3) Trasferimento risorse
 - Invia tutte le risorse al successore
- 4) Cleanup Finale
 - Greatful stop di tutte le strutture del nodo

Deploy

Tracing, testing e Dimostrativo

Deploy per tracing analysis locale

Debugging

Tracing Analysis Locale

Scopo:

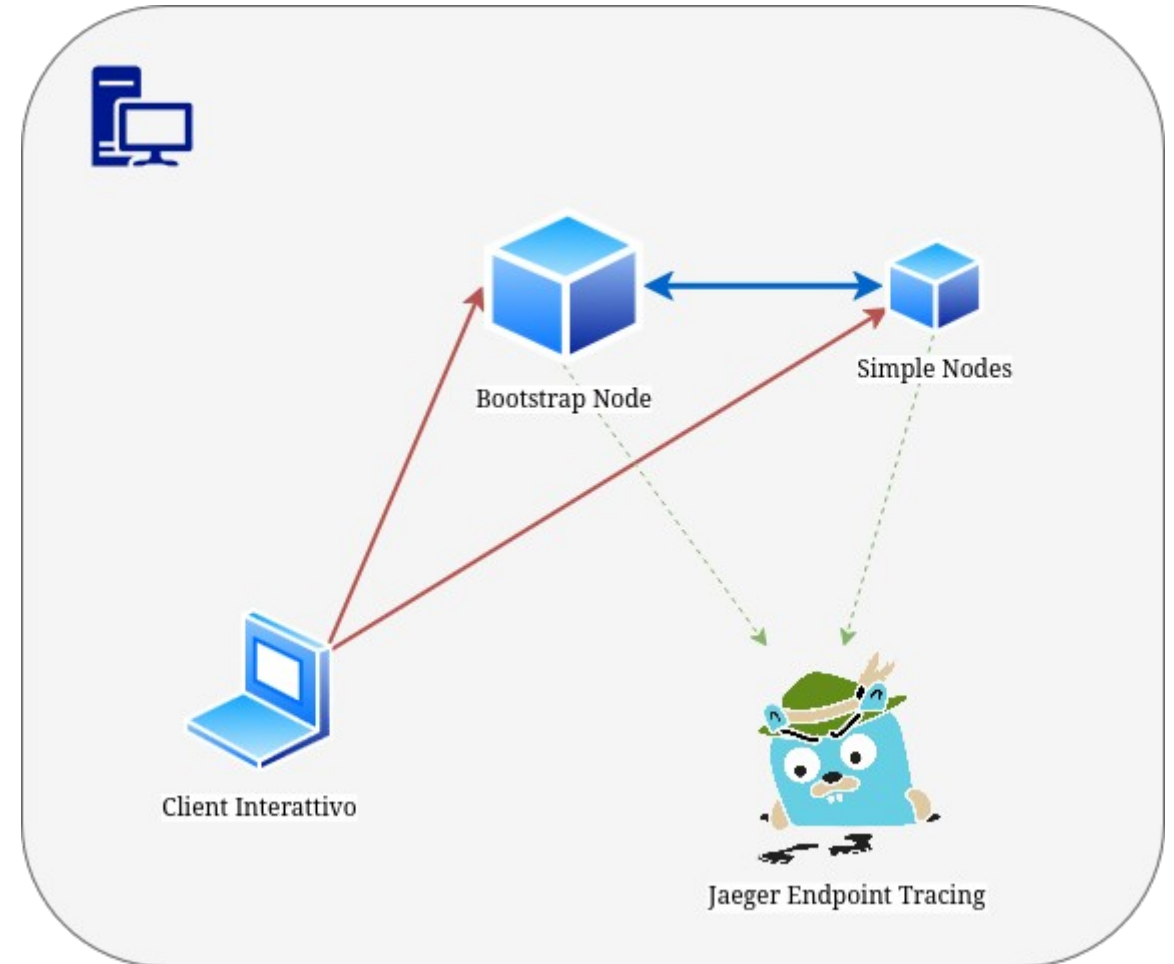
- Sviluppo, debug e analisi del traffico gRPC

Caratteristiche:

- Tutti i container in una sola macchina locale
- Rete virtuale Docker (koordenet) isolata
- Bootstrap statico

Uso:

- Docker Compose
- <http://localhost:16686> (Jaeger UI)



Deploy di Testing su singola EC2

Testing

Testing Deploy su singola EC2

Scopo:

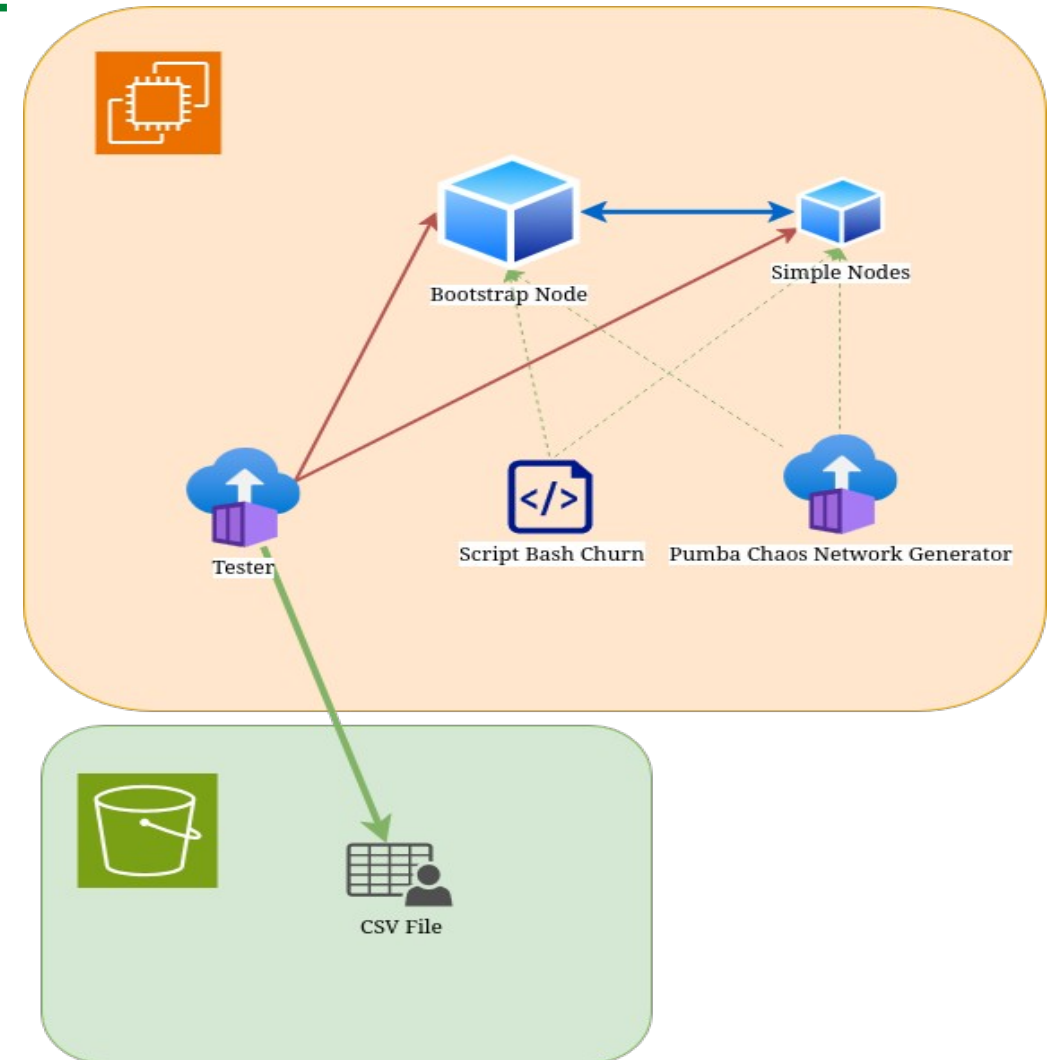
- test realistico di latenza, perdita e churn

Caratteristiche:

- Tutti i container in una sola macchina EC2
- Script bash per Churn
- Pumba container per chaos network
- Bootstrap statico

Uso:

- Tramite script `deploy_test.sh` che usa CloudFormation
- Prendere il csv all'interno del bucket s3 specificato



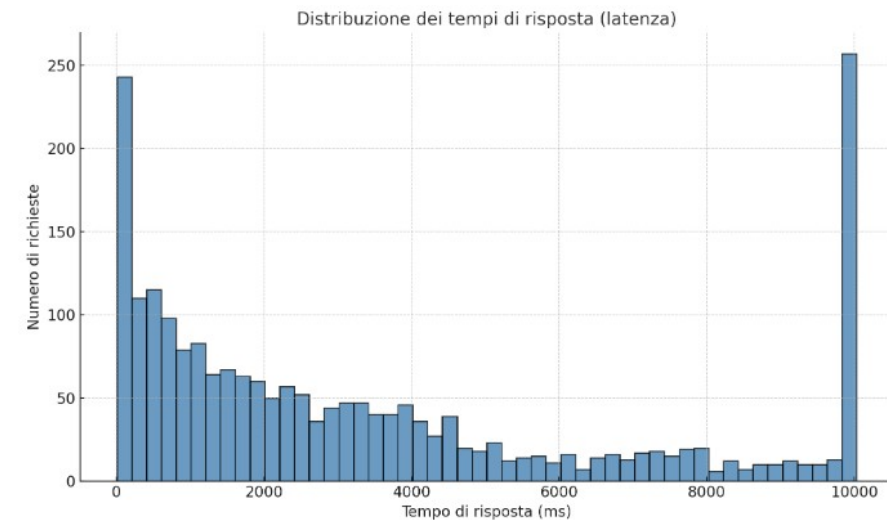
Esempio di risultati

Latenza:

- Delay : 100ms
- Jitter: 50ms
- Loss: 0.1%

Churn:

- Tasso: 0.5
- Probabilità di Join: 50%
- Probabilità di Leave: 50%



Deploy dimostrativo multistanza EC2

Demo dimostrativa

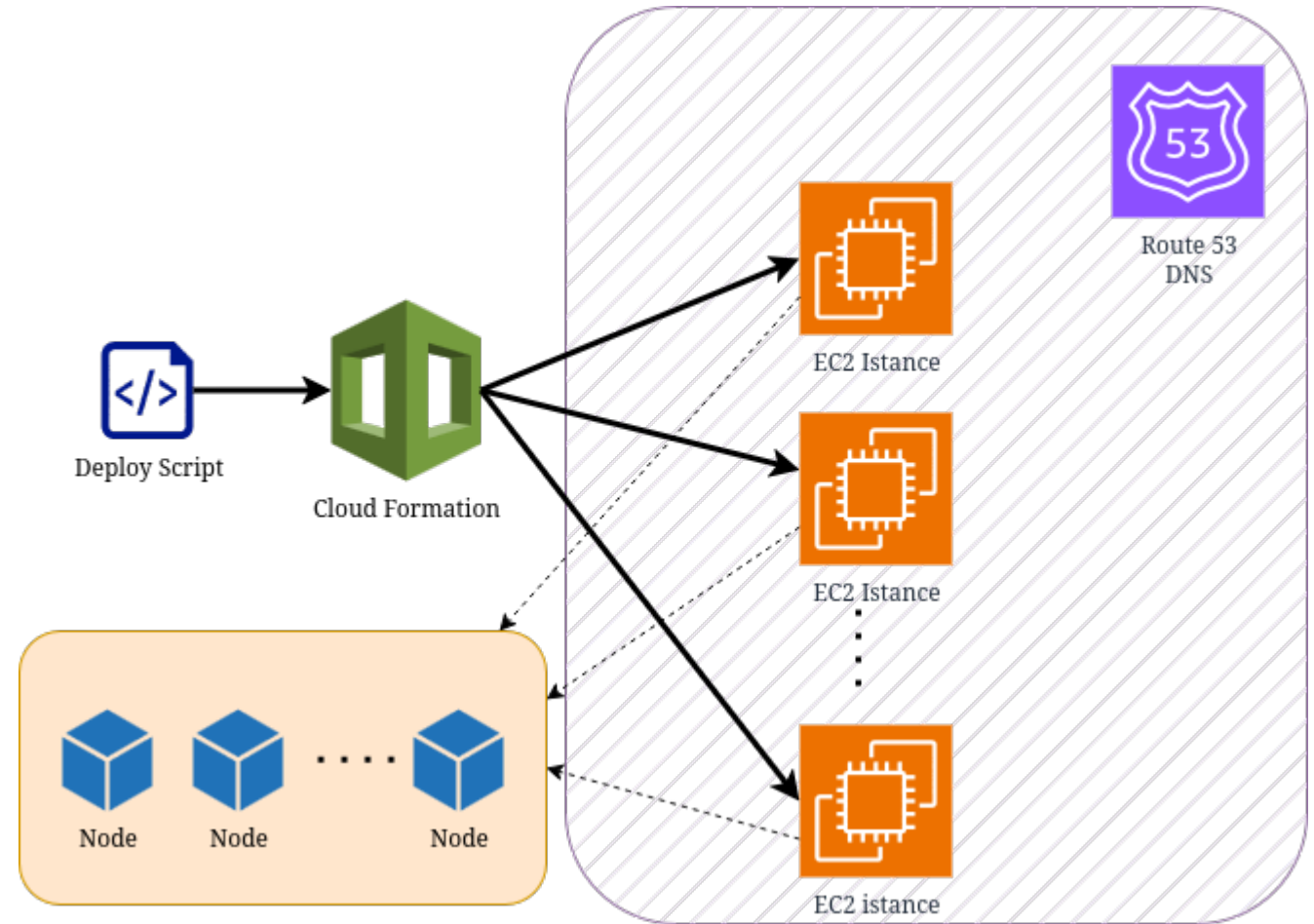
Deploy dimostrativo multistanza EC2

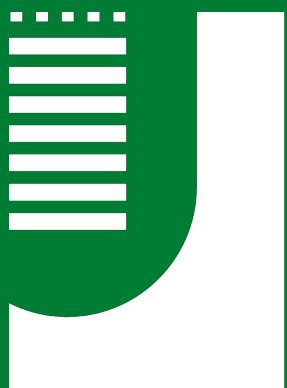
Scopo:

- rappresentare un ambiente production-like

Caratteristiche:

- Più istanze EC2, ciascuna con più container DHT
- DNS dinamico via Route53 **SRV** records
- Client interattivo locale connesso via IP pubblico a un solo nodo simulando un real client che ignora la struttura interna





TOR VERGATA
UNIVERSITÀ DEGLI STUDI DI ROMA