Lezione n.5

DHT: CHORD

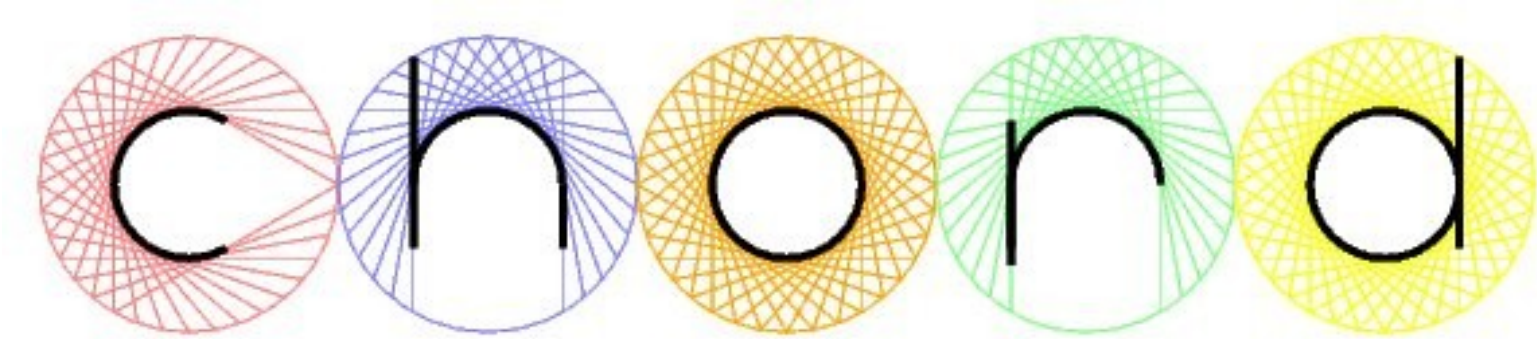
Laura Ricci

18/10/2013

RIASSUNTO DELLA PRESENTAZIONE

- Chord: idee generali
- Topologia
- Routing
- Auto Organizzazione
  - Arrivo nuovi nodi
  - Partenza volontaria
  - Faults

CHORD: INTRODUZIONE



- Materiale didattico sul libro, fonte:

Ion Stoica, Robert Morris, David Liben-Nowell, David R. Karger, M. Frans Kaashoek, Frank Dabek, Hari Balakrishnan, Chord: A Scalable Peer-to-peer Lookup Protocol for Internet Applications. IEEE/ACM Transactions on Networking

•sviluppato nel 2001 da un gruppo composto da ricercatori del MIT, Università della California

CHORD: INTRODUZIONE

- Basato su pochi semplici concetti
  - Facilità di comprensione e di implementazione
  - Eleganza
  - Possibilità di definire ottimizzazioni
- Caratteristiche Principali:
  - Routing
    - Spazio Logico degli indirizzi piatto: gli indirizzi sono identificatori di m-bits invece che indirizzi IP
    - Routing efficiente: log(N) hops con alta probabilità, se N è il numero totale di nodi del sistema.
    - Dimensione delle tabelle di routing log(N) con alta probabilità
  - Auto-organizzazione
    - Gestisce inserzione di nuovi nodi, ritiri volontari dal sistema, fallimenti

CHORD: APPLICAZIONI

- Diversi sistemi proposti a livello accademico

- Sistemi commerciali:
  - Amazon Dynamo: Una variante di Chord è alla base della definizione del sistema di storage distribuito definito per il cloud di Amazon. Obiettivi
    - bassa latenza, modifiche al routing base di Chord
    - Alta disponibilità
    - Adattabile all'esigenze dell'utente

  - Apache Cassandra:
    - Sviluppato originariamente da Facebook
    - Variante di Dynamo, sempre basata su Chord
    - Utilizzata anche da Digg e WebEx di Cisco

CHORD: LA TOPOLOGIA

- Hash-table storage
  - put (key, value) per inserire dati in Chord
  - value = get (key) per ricercare dati

- Generazione degli identificatori mediante Hash. Si utilizza SHA-1 (Secure Hash Standard)

- Ad ogni nodo(host) viene associato un identificatore ID
  - id = sha-1 (indirizzo IP, porta)

- Ad ogni dato viene associata una chiave key
  - key = sha-1 (dato)

- Chiavi ed identificatori sono mappati nello stesso spazio degli indirizzi

CHORD: SHA1 E CONSISTENT HASHING

- Si utilizza SHA1 per assegnare identificatori unici a chiavi e nodi
  - la probabilità che due nodi diversi o due chiavi diverse generino lo stesso identificatore deve essere trascurabile
  - per questo è necessario utilizzare identificatori sufficientemente lunghi
    - Ad esempio 160-bit ($0 \leq$ identificatore $\leq 2^{160}$)
- L'associazione delle chiavi ai nodi avviene mediante consistent hashing che garantisce
  - bilanciamento del carico: le chiavi vengono distribuite uniformemente tra i nodi
  - L'inserimento/ eliminazione di un nodo comporta lo spostamento di un numero limitato di chiavi. Se inserisco l'N-esimo nodo sposto 1/N chiavi

CHORD: LA TOPOLOGIA

- Ipotesi: consideriamo identificatori di m bits, $[0, 2^m-1]$
- Si stabilisce un ordinamento tra gli identificatori, in base al loro valore numerico. Il successore di 2^m-1 è 0.
- L'ordinamento può essere rappresentato mediante un anello (Chord Ring)
- Algoritmo di Mapping

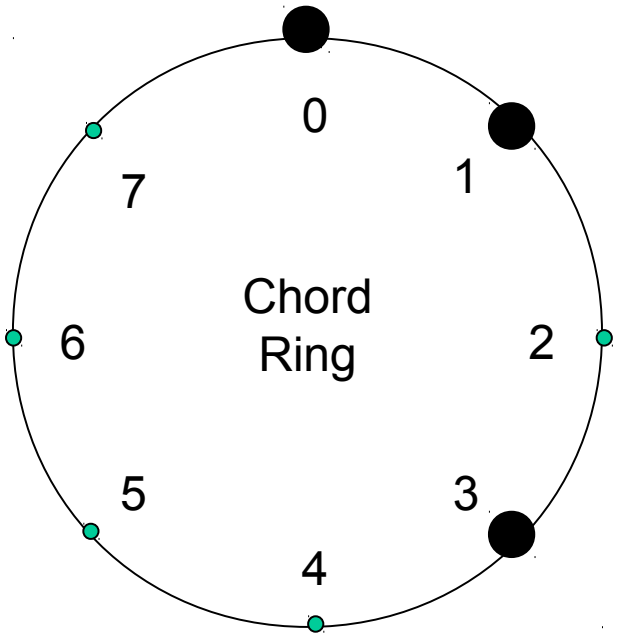
  Una chiave di valore K viene assegnata al primo nodo n dell'anello il cui identificatore ID risulta maggiore o uguale a K.

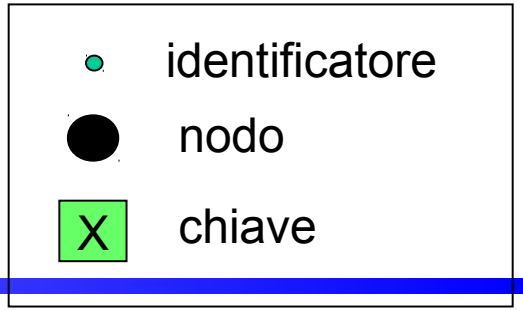
  n(K)= successor(K)

- n è il primo nodo individuato partendo da K e proseguendo sull'anello Chord in senso orario.

CHORD: LA TOPOLOGIA

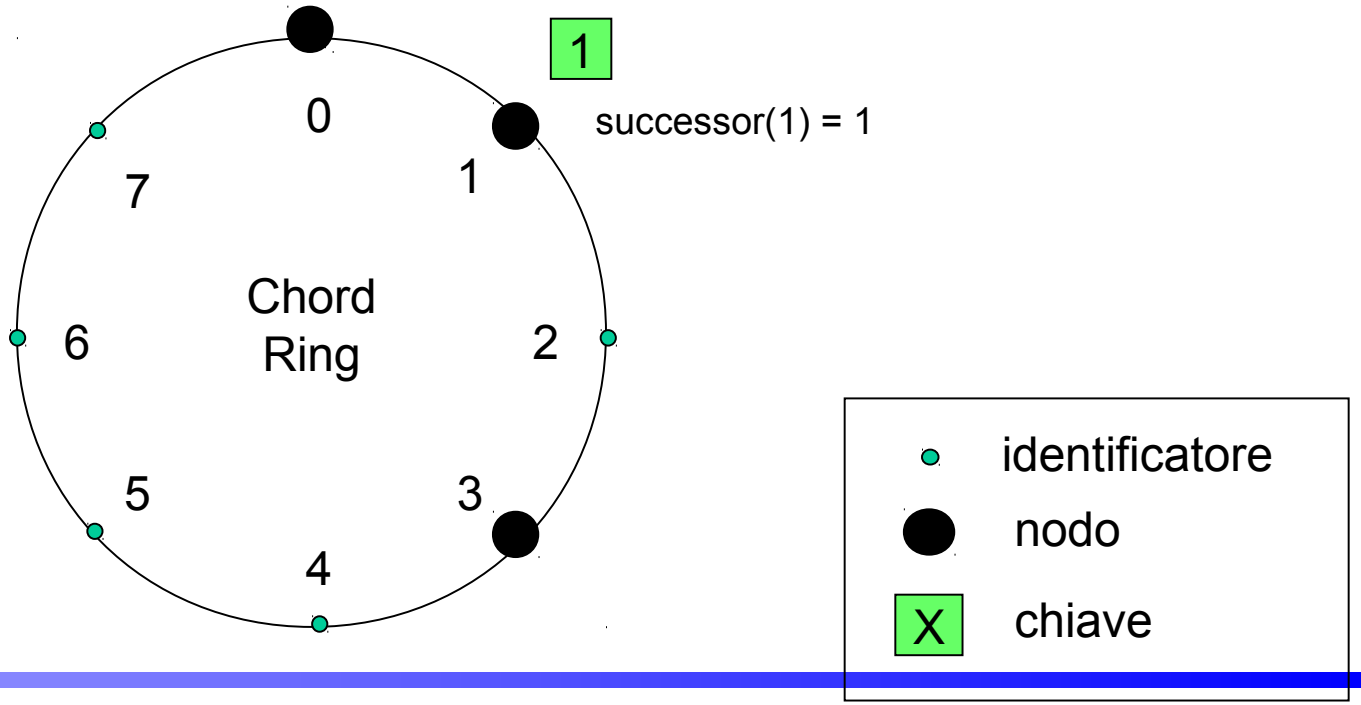
- L'anello contiene chiavi ed IDs, i.e., tutta l'aritmetica è modulo 2^{160}
- La chiave ed il valore ad essa associato sono gestiti dal nodo successivo della chiave in senso orario sull'anello Chord





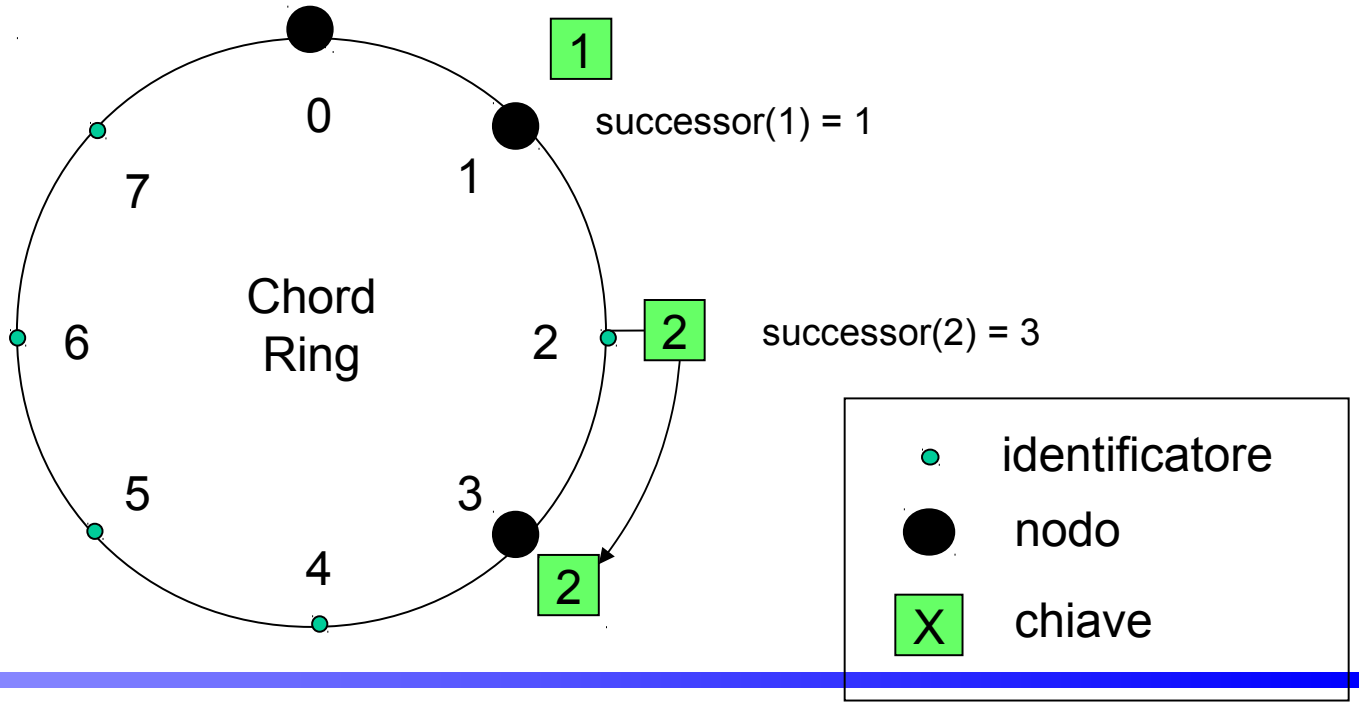
CHORD: LA TOPOLOGIA

- L'anello contiene chiavi ed IDs, i.e., tutta l'aritmetica è modulo 2^{160}
- La coppia (chiave, valore) è gestita dal nodo successivo della chiave in senso orario sull'anello Chord



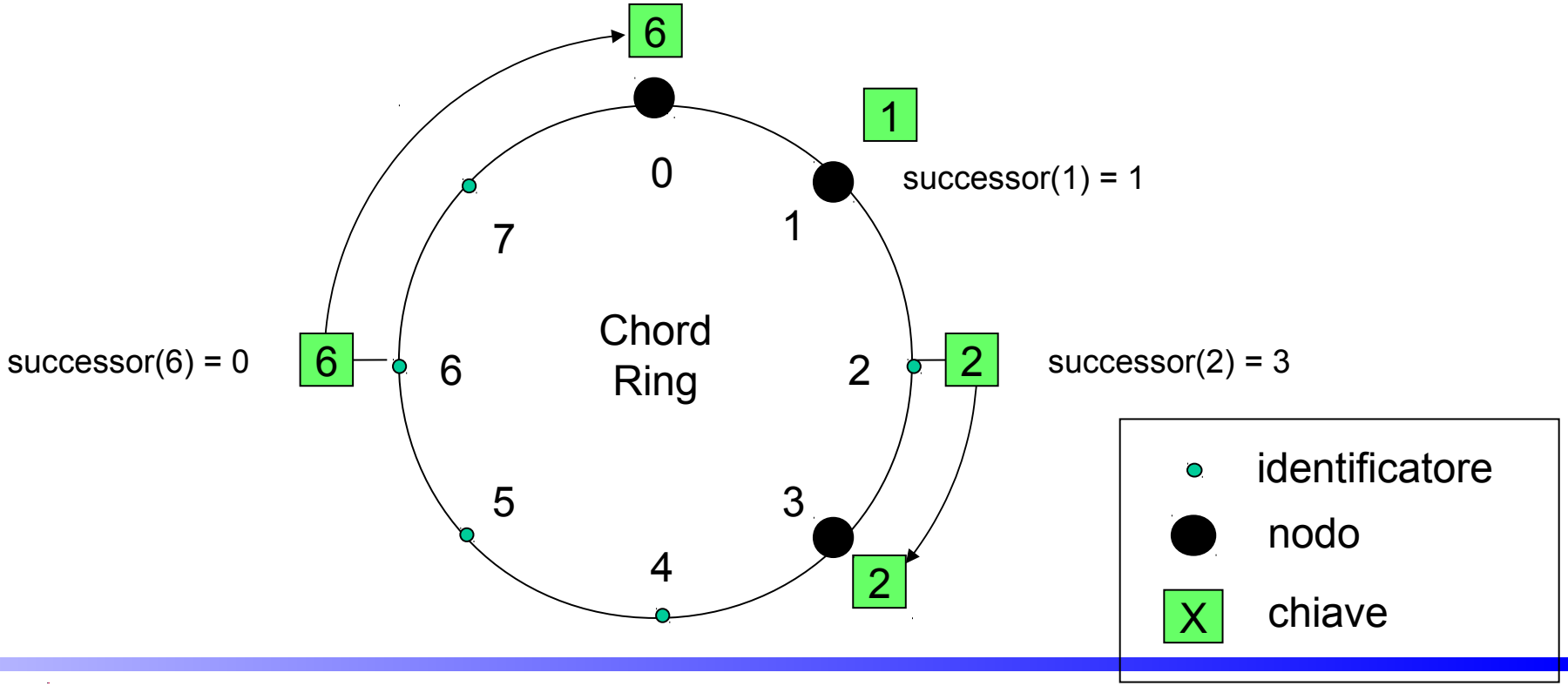
CHORD: LA TOPOLOGIA

- L'anello contiene chiavi ed IDs, i.e., tutta l'aritmetica è modulo 2^{160}
- La coppia (chiave, valore) è gestita dal nodo successivo della chiave in senso orario sull'anello Chord



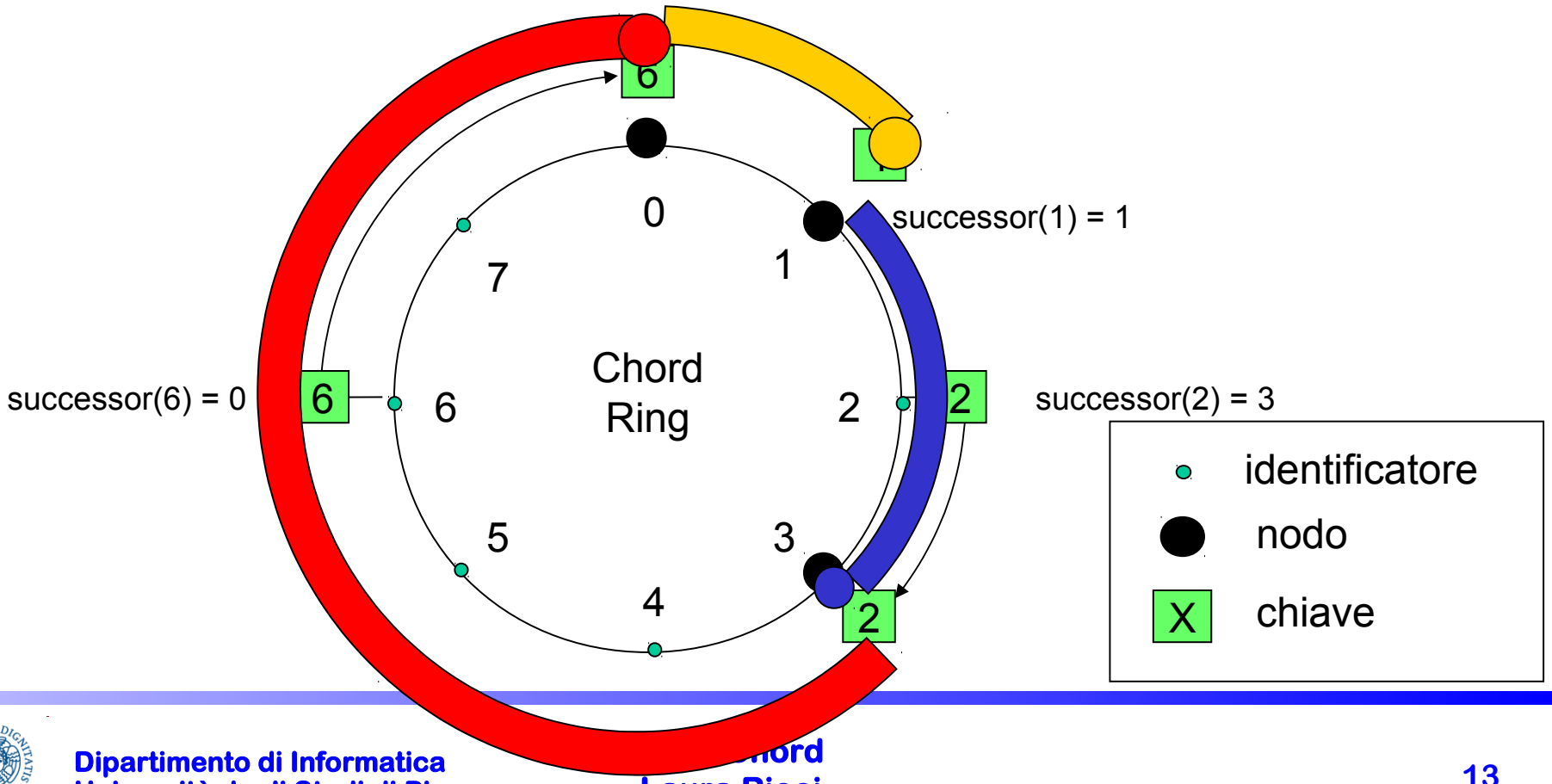
CHORD: LA TOPOLOGIA

- L'anello contiene chiavi ed IDs, i.e., tutta l'aritmetica è modulo 2^{160}
- La coppia (chiave, valore) è gestita dal nodo successivo della chiave in senso orario sull'anello Chord



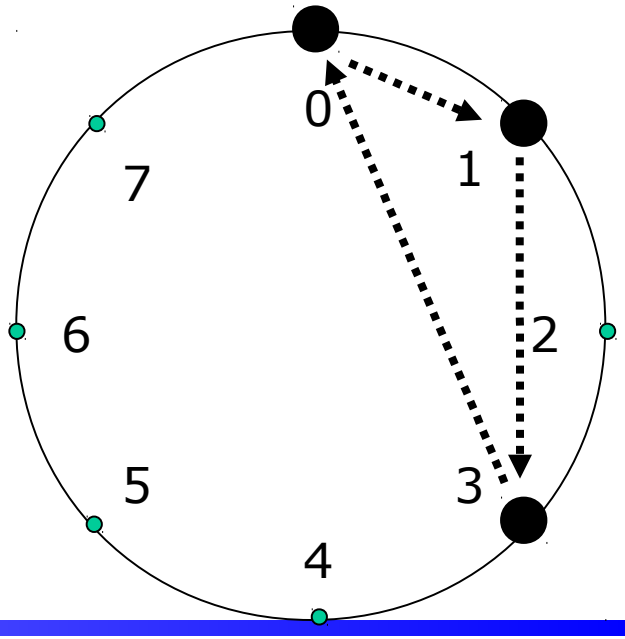
CHORD: LA TOPOLOGIA

- L'anello contiene chiavi ed IDs, i.e., tutta l'aritmetica è modulo 2^{160}
- La coppia (chiave, valore) è gestita dal nodo successivo della chiave in senso orario sull'anello Chord



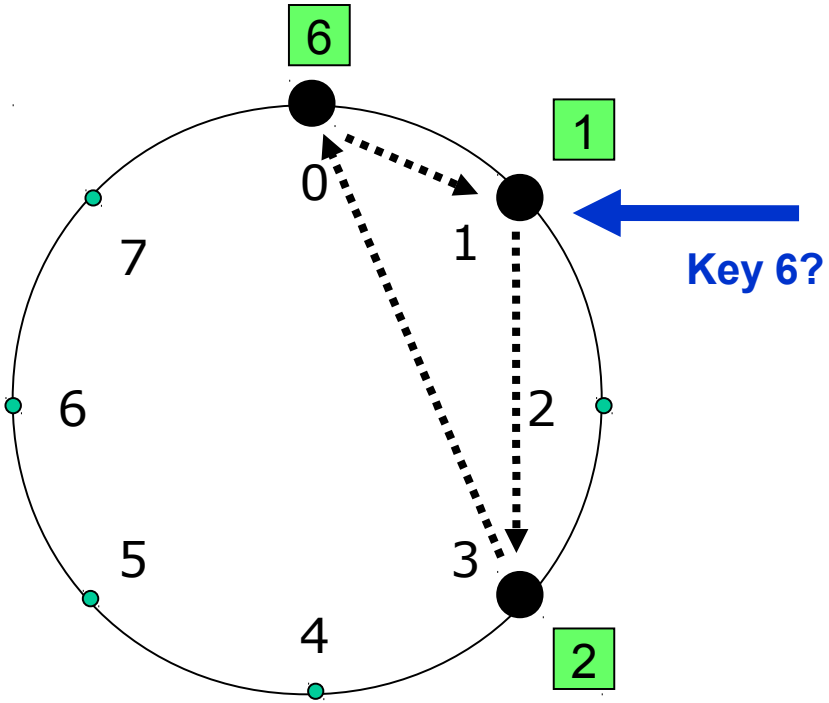
CHORD: L'OVERLAY

- L'overlay è determinato dai links stabiliti tra i nodi dell'anello
  - i links memorizzati in ogni nodo rappresentano la conoscenza che un nodo possiede degli altri nodi dell'anello
  - sono memorizzati nella Routing Table di ogni nodo
- La topologia più semplice: lista circolare
  - ogni nodo possiede un link verso il nodo successivo, in senso orario



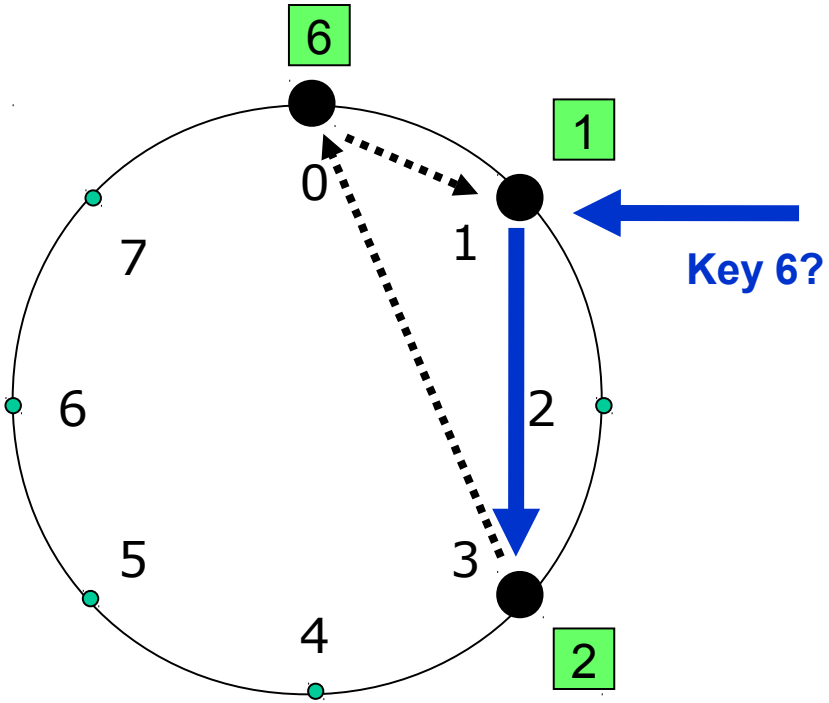
CHORD: IL ROUTING

- Un semplice algoritmo di Routing:
  - ogni nodo possiede solo un link verso il suo successore
  - inoltra la query per la chiave x al suo successore finchè non individua n =successor(x)
  - n restituisce i risultati della query
- Vantaggi:
  - semplice
  - tabelle di routing O(1)
- Svantaggi:
  - Routing $O(\frac{1}{2} \times n)$, lineare
  - Il fallimento di un nodo interrompe i collegamenti sull'anello



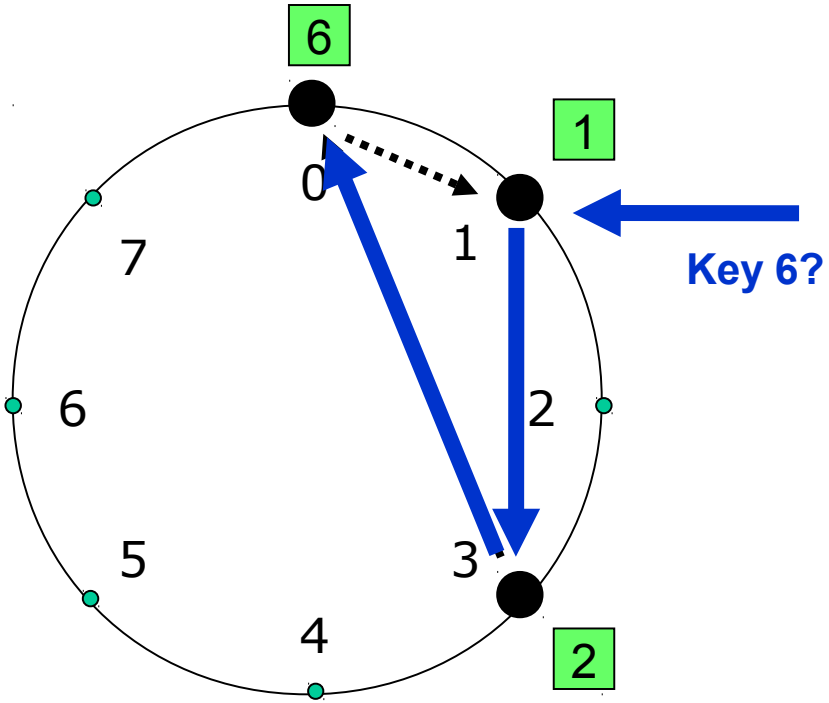
CHORD: IL ROUTING

- Un semplice algoritmo di Routing:
  - Ogni nodo possiede solo un link verso il suo successore
  - Inoltra la query per la chiave x al suo successore finchè non individua n=successor(x)
  - n restituisce i risultati della query
- Vantaggi:
  - semplice
  - Tabella di routing O(1)
- Svantaggi:
  - Routing $O(\frac{1}{2} * n)$, lineare
  - Il fallimento di un nodo interrompe i collegamenti sull'anello



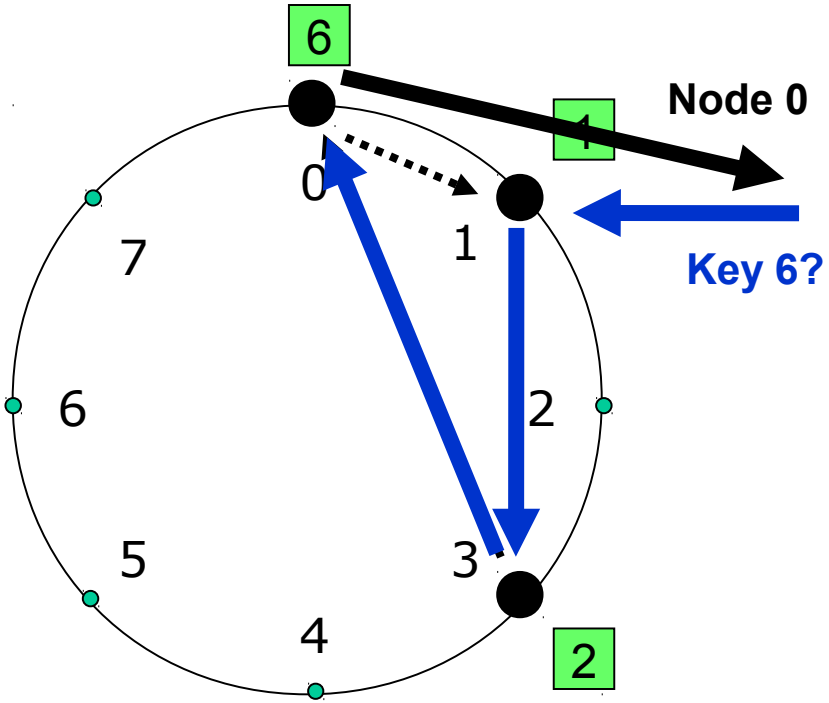
CHORD: IL ROUTING

- Un semplice algoritmo di Routing:
  - Ogni nodo possiede solo un link verso il suo successore
  - Inoltra la query per la chiave x al suo successore finchè non individua n=successor(x)
  - Restituisce i risultati della query
- Vantaggi:
  - Semplice
  - Tabelle di routing O(1)
- Svantaggi:
  - Routing $O(\frac{1}{2} * n)$, lineare
  - Il fallimento di un nodo interrompe i collegamenti sull'anello



CHORD: IL ROUTING

- Un semplice algoritmo di Routing:
  - Ogni nodo possiede solo un link verso il suo successore
  - Inoltra la query per la chiave x al suo successore finchè non individua n=successor(x)
  - Restituisce i risultati della query
- Vantaggi:
  - Semplice
  - Tabelle di Routing O(1)
- Svantaggi:
  - Routing $O(\frac{1}{2} * n)$, lineare
  - Il fallimento di un nodo interrompe i collegamenti sull'anello



CHORD: IL ROUTING

- Ogni nodo memorizza i links a z vicini

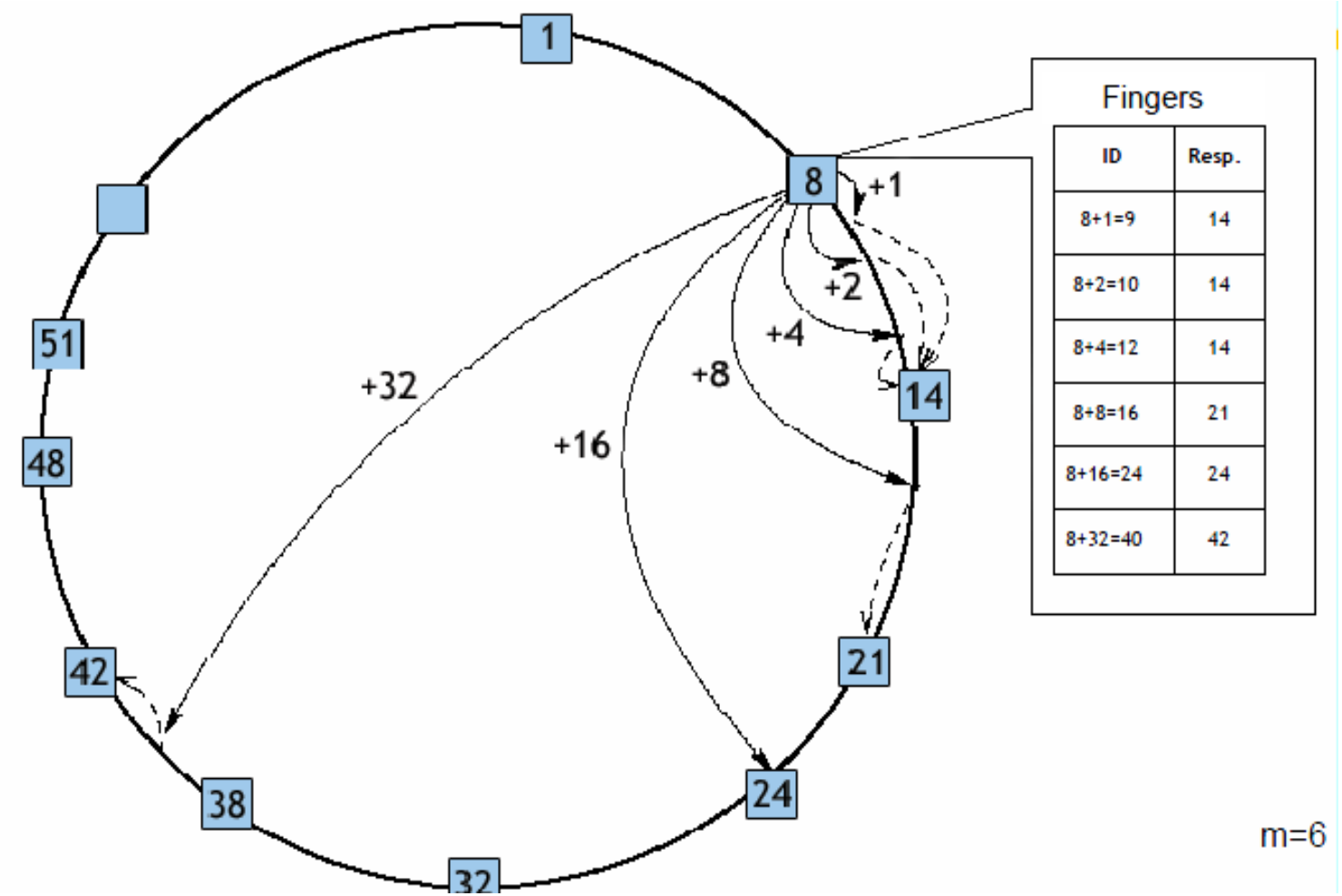
- Se z = n si ottiene un mesh completo
  - routing: O(1),
  - dimensione delle tabelle di routing: O(n), scalabilità limitata

- Compromesso: ogni nodo memorizza diversi di links verso alcuni nodi vicini (nell'anello) e solo alcuni verso nodi lontani
  - numero limitato di links per ogni nodo
  - routing accurato in prossimità di un nodo, più approssimato verso nodi lontani
  - algoritmo di routing:
    - inoltrare la query per una chiave k al predecessore di k più lontano, conosciuto

CHORD: IL ROUTING

- Definiamo distanza tra due identificatori I_1 ed I_2 dell'anello Chord come il numero di identificatori compresi tra I_1 ed I_2
- Idea base per la costruzione della tabella di routing (finger table)
  - ogni nodo di identificatore x appartenente all'anello Chord, conosce un insieme di al più m nodi (in realtà sono meno) costituito da nodi caratterizzati da identificatori la cui distanza da x cresce esponenzialmente
  - Per considerare nodi a distanze esponenzialmente crescenti, si considerano i nodi che si trovano a distanza 2^i da x, dove $0 \leq i \leq m-1$
  - Inoltre ogni nodo conosce il suo successore ed il suo predecessore sull'anello

CHORD: LE FINGER TABLES



CHORD: LE FINGER TABLES

Ogni nodo utilizza una finger table (tabella di routing)

- se m è il numero di bits utilizzati per gli identificatori, la tabella contiene al massimo m links che contengono riferimenti ad altri nodi Chord
- sul nodo n: l'entrata finger[i] punta a successor($n + 2^{i-1}$), $1 \leq i \leq m$

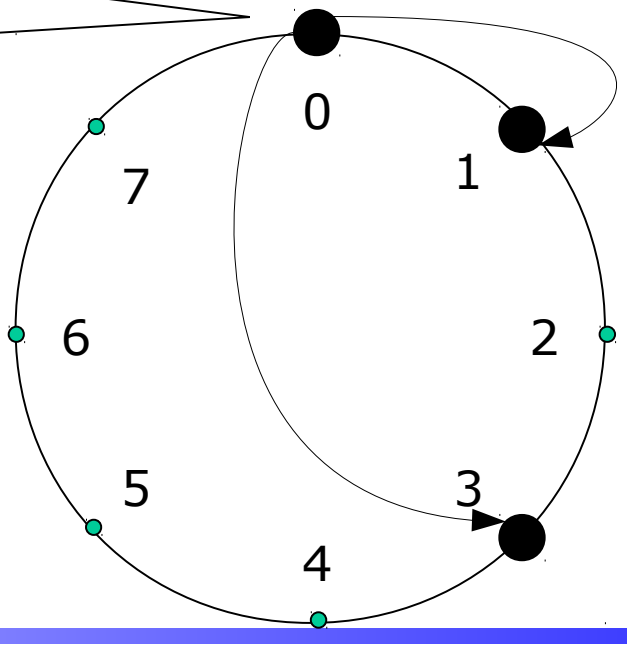
Strutture Dati del Nodo 0

finger table

i	target	link.
1	1	1
2	2	3
3	4	0

keys

6



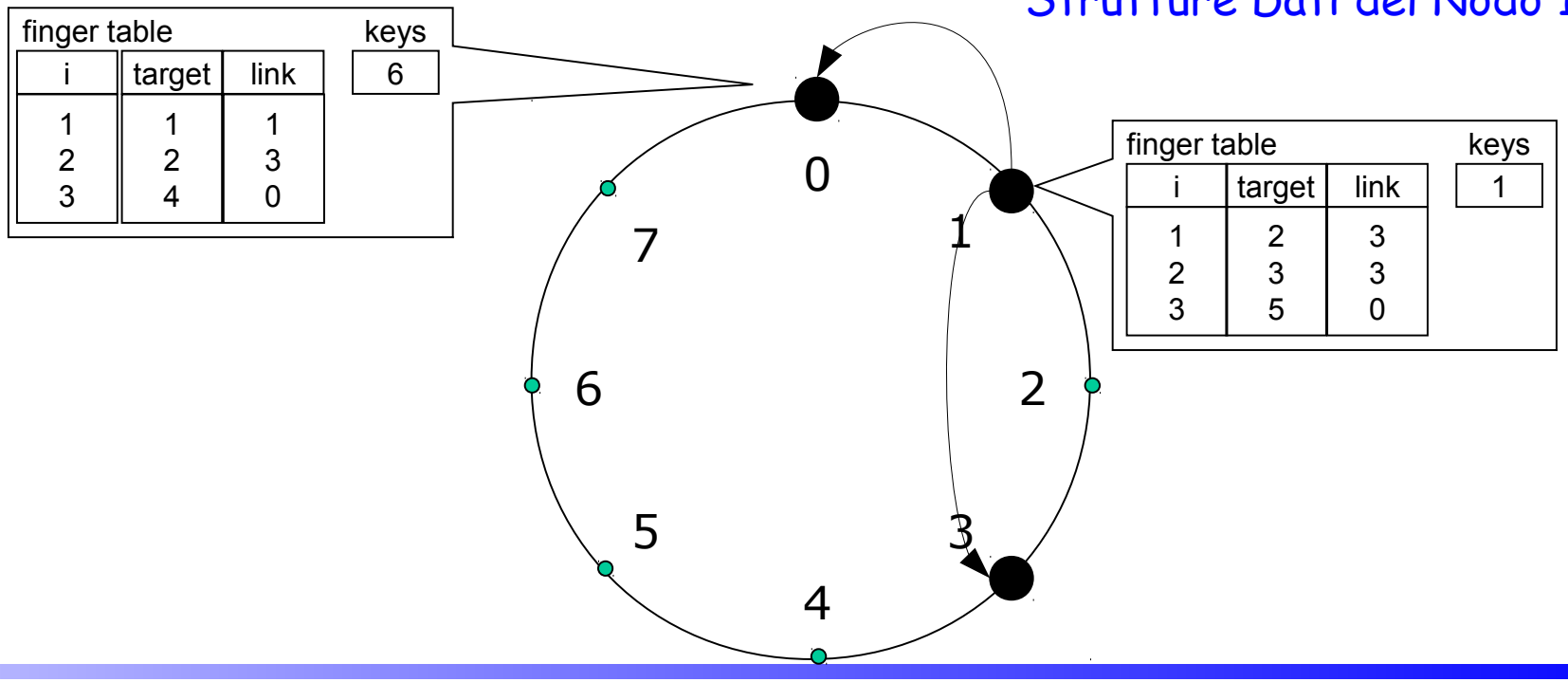
CHORD: LE FINGER TABLES

Routing Table: ogni nodo utilizza una finger table

- se m è il numero di bits utilizzati per gli identificatori, la tabella contiene al massimo m links che contengono riferimenti ad altri nodi Chord
- sul nodo n: l'entrata finger[i] punta a successor($n + 2^{i-1}$), $1 \leq i \leq m$

Strutture Dati del Nodo 0

Strutture Dati del Nodo 1



CHORD: LE FINGER TABLES

Routing Table: ogni nodo utilizza una finger table

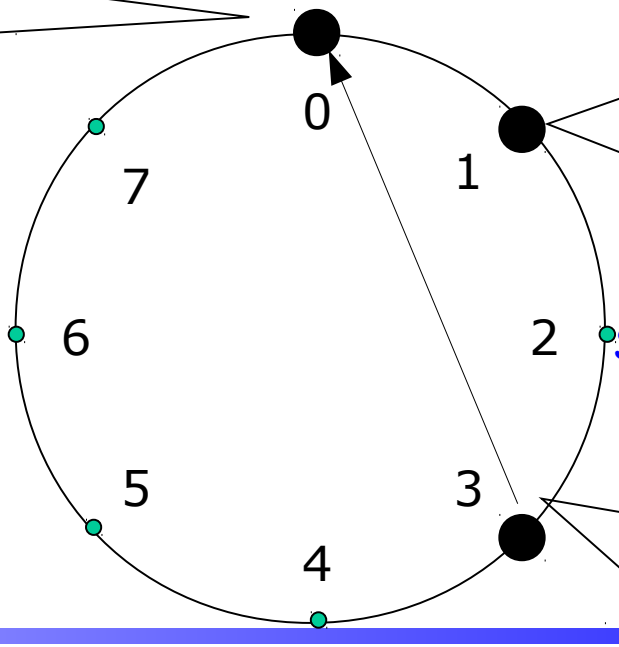
- se m è il numero di bits utilizzati per gli identificatori, la tabella contiene al massimo m links che contengono riferimenti ad altri nodi Chord
- sul nodo n: l'entrata finger[i] punta a successor($n + 2^{i-1}$), $1 \leq i \leq m$

Strutture Dati del Nodo 0

finger table

i	target	link.
1	1	1
2	2	3
3	4	0

keys: 6



Strutture Dati del Nodo 1

finger table

i	target	link.
1	2	3
2	3	3
3	5	0

keys: 1

Strutture Dati del Nodo 3

finger table

i	target	link.
1	4	0
2	5	0
3	7	0

keys: 2

CHORD: STRUTTURE DATI DI UN NODO

- I peer sono organizzati secondo un anello logico
- Ogni nodo mantiene
  - Alcune strutture dati necessarie per il routing
    - Finger table:
      - possiedono un numero di entrate logaritmico rispetto al numero di identificatori della rete,
      - solo un sottoinsieme di queste contiene valori diversi
      - overlay = mesh di grado logaritmico tra i nodi del'anello Chord
    - Puntatore al nodo successore ed al nodo predecessore sull'anello
  - L'elenco delle chiavi che sono state mappate su quel nodo
  - Un insieme di puntatori a nodi successivi necessari per garantire la consistenza della rete in caso di join/leave dinamici dei nodi

CHORD: IL ROUTING

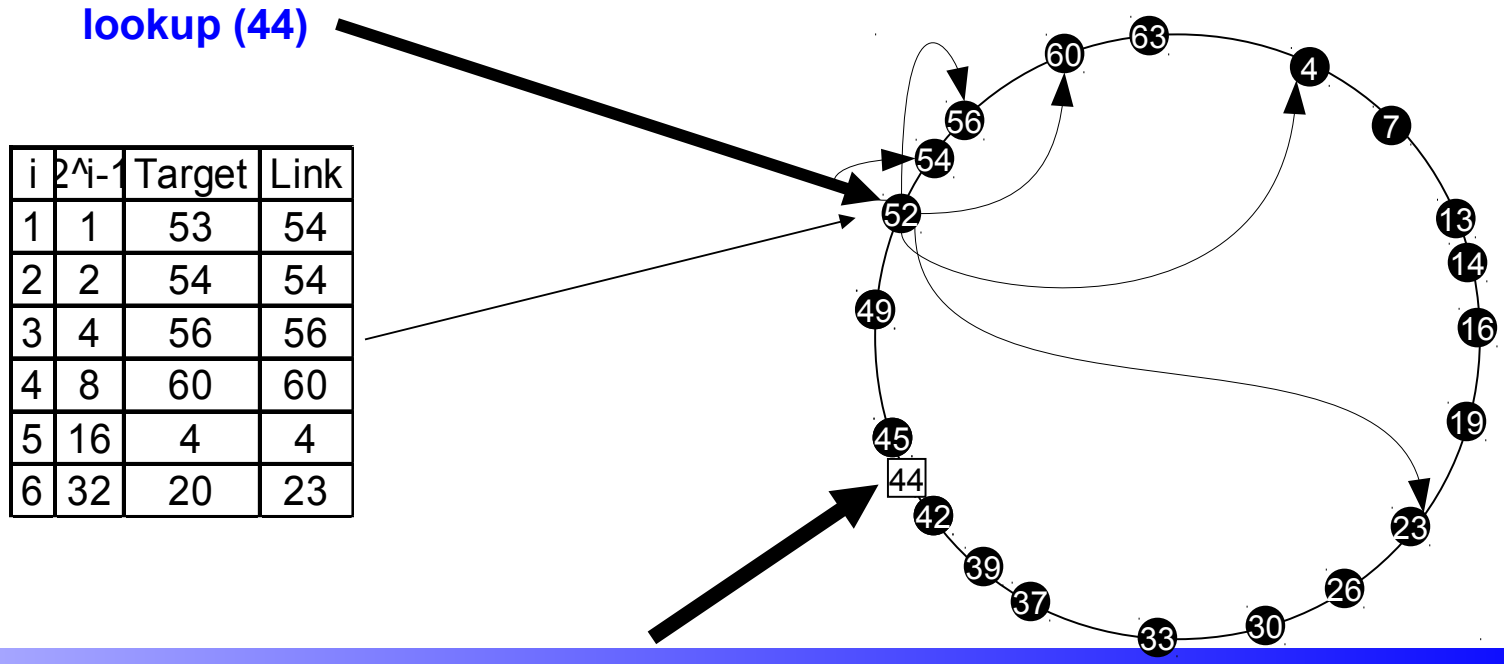
- Le operazioni che possono essere richiesta alla DHT da un qualsiasi nodo della rete sono le seguenti
  - PUT (Key, Value)
  - GET (Key)
- In entrambe i casi il nodo fornisce la chiave dell'informazione che vuole memorizzare/ricercare
- Key-Based Routing: ogni nodo che riceve la chiave la instrada ad un altro nodo della rete scelto nella propria finger table. La scelta del nodo è guidata dal valore della chiave.
- La chiave viene instradata fino a che non si individua il successore della chiave sull'anello

CHORD: IL ROUTING

- Algoritmo di Routing : ogni nodo n propaga una query per la chiave k al finger più distante che precede k, in senso orario
- La propagazione continua fino al nodo n tale che

  n = predecessore(k) e successore(n) = successore(k)

in questo caso il successore di n possiede la chiave

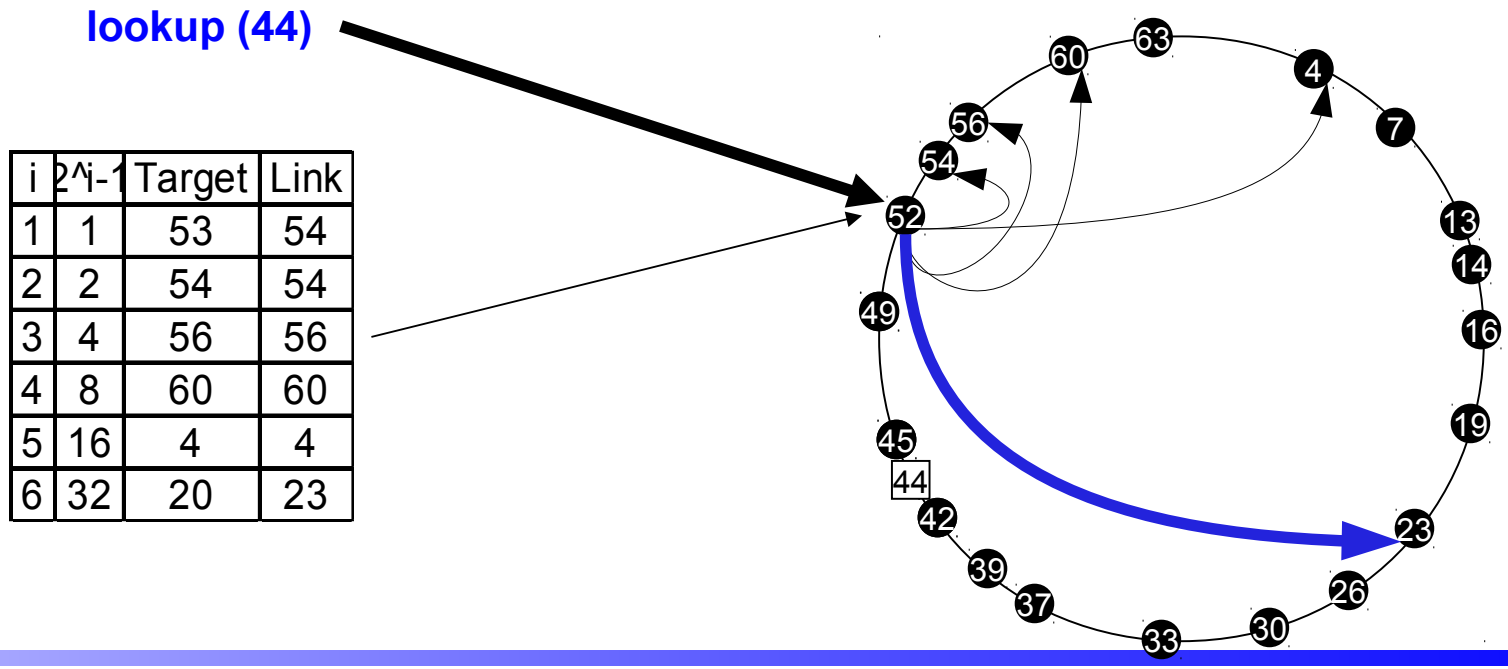


CHORD: IL ROUTING

- Algoritmo di Routing di Chord: ogni nodo n propaga la query per la chiave k al finger più distante che precede k, in senso orario
- La propagazione continua fino al nodo n tale che

  n = predecessore(k) e successore(n) = successore(k)

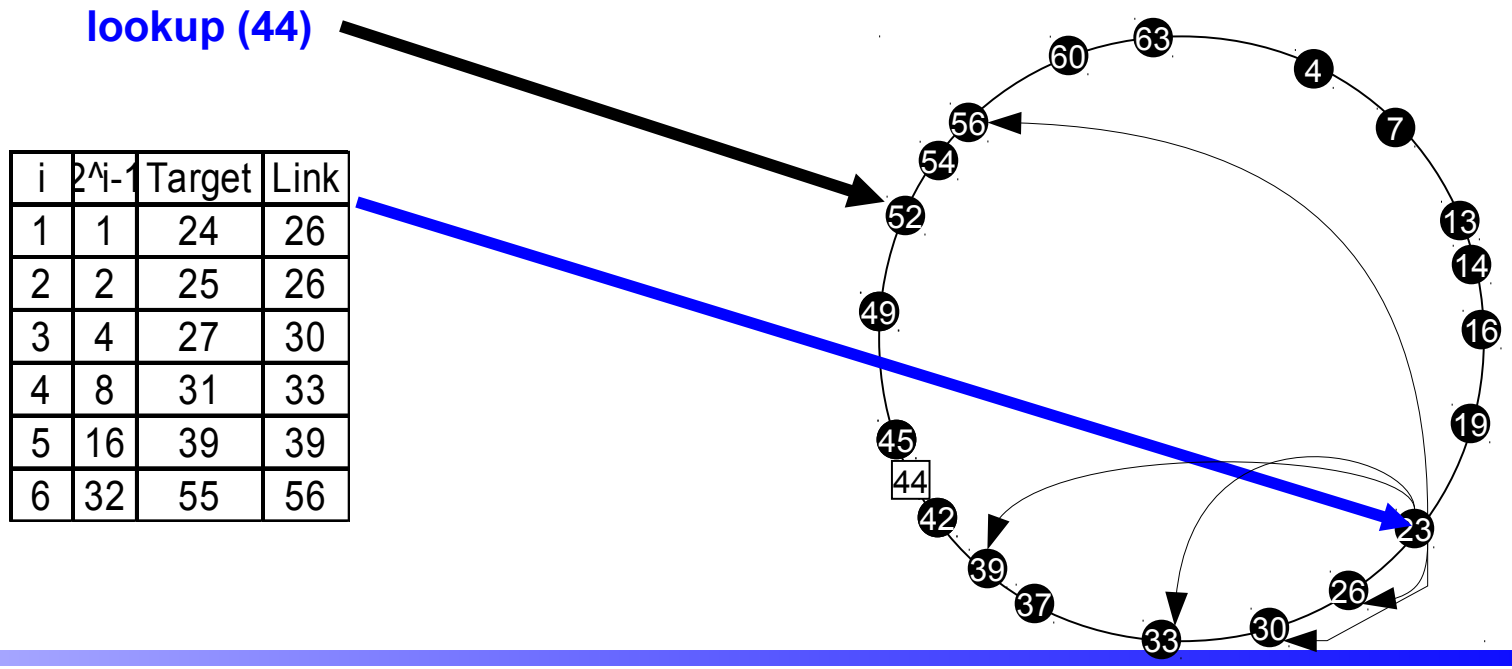
  in questo caso il successore di n prossiede la chiave



CHORD: IL ROUTING

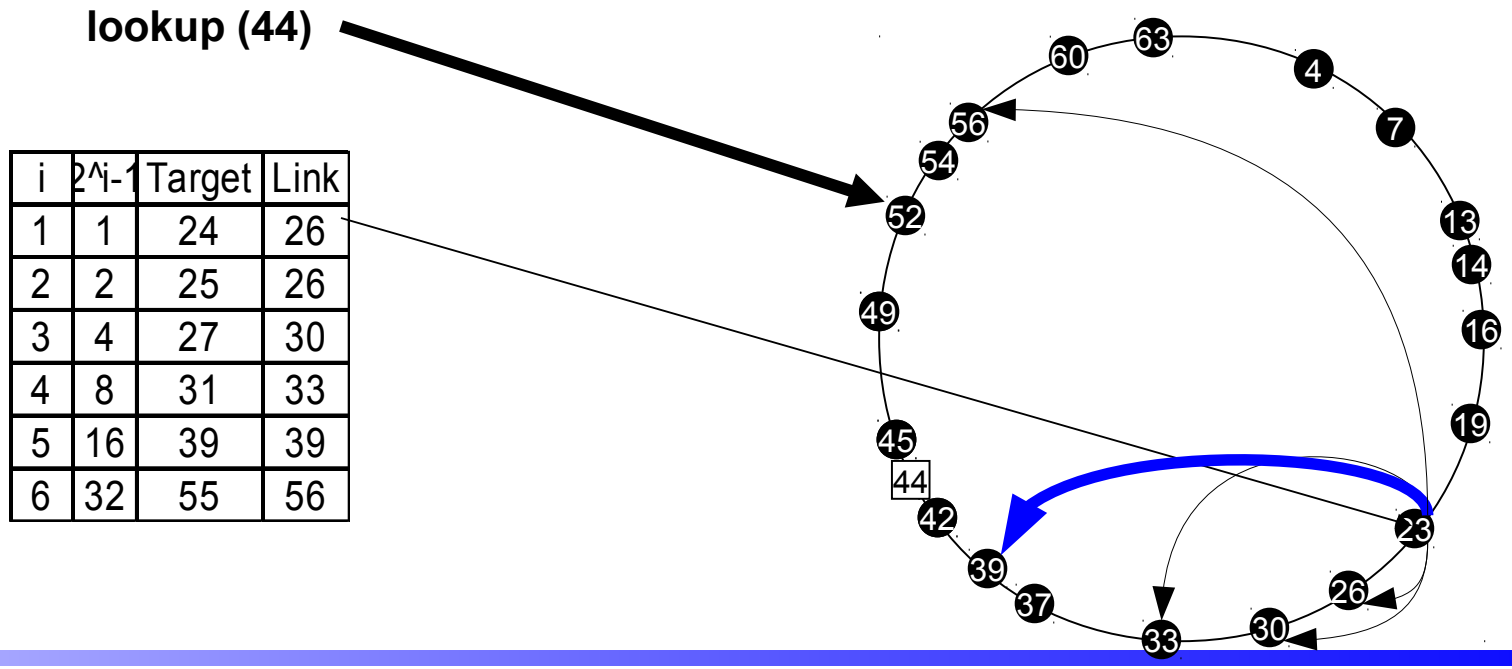
- Algoritmo di Routing di Chord: ogni nodo n propaga la query per la chiave k al finger più distante che precede k, in senso orario
- La propagazione continua fino al nodo n tale che n = predecessore(k) e successore(n) = successore(k)

in questo caso il successore di n possiede la chiave



CHORD: IL ROUTING

- Algoritmo di Routing di Chord: ogni nodo n propaga la query per la chiave k al finger più distante che precede k, in senso orario
- La propagazione continua fino al nodo n tale che n = predecessore(k) e successore(n) = successore(k) in questo caso il successore di n possiede la chiave

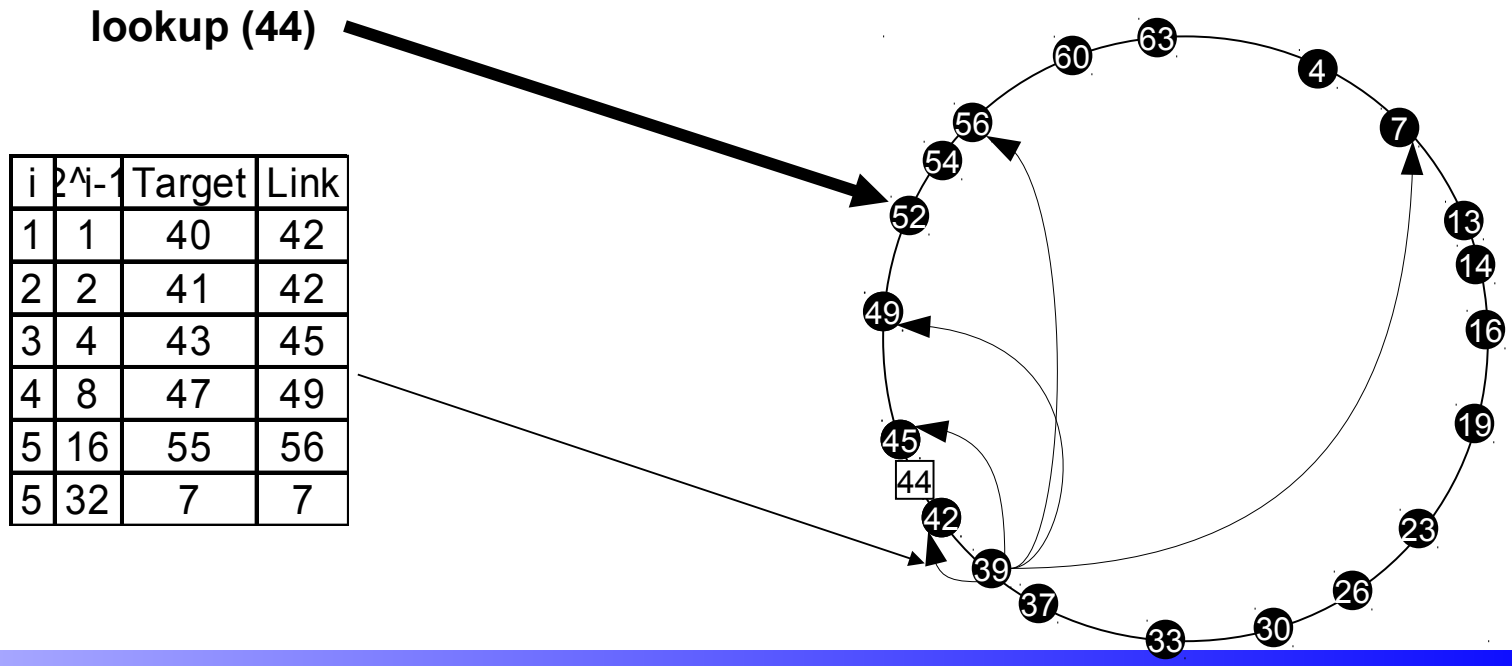


CHORD: IL ROUTING

- Algoritmo di Routing di Chord: ogni nodo n propaga la query per la chiave k al finger più distante che precede k, in senso orario
- La propagazione continua fino al nodo n tale che

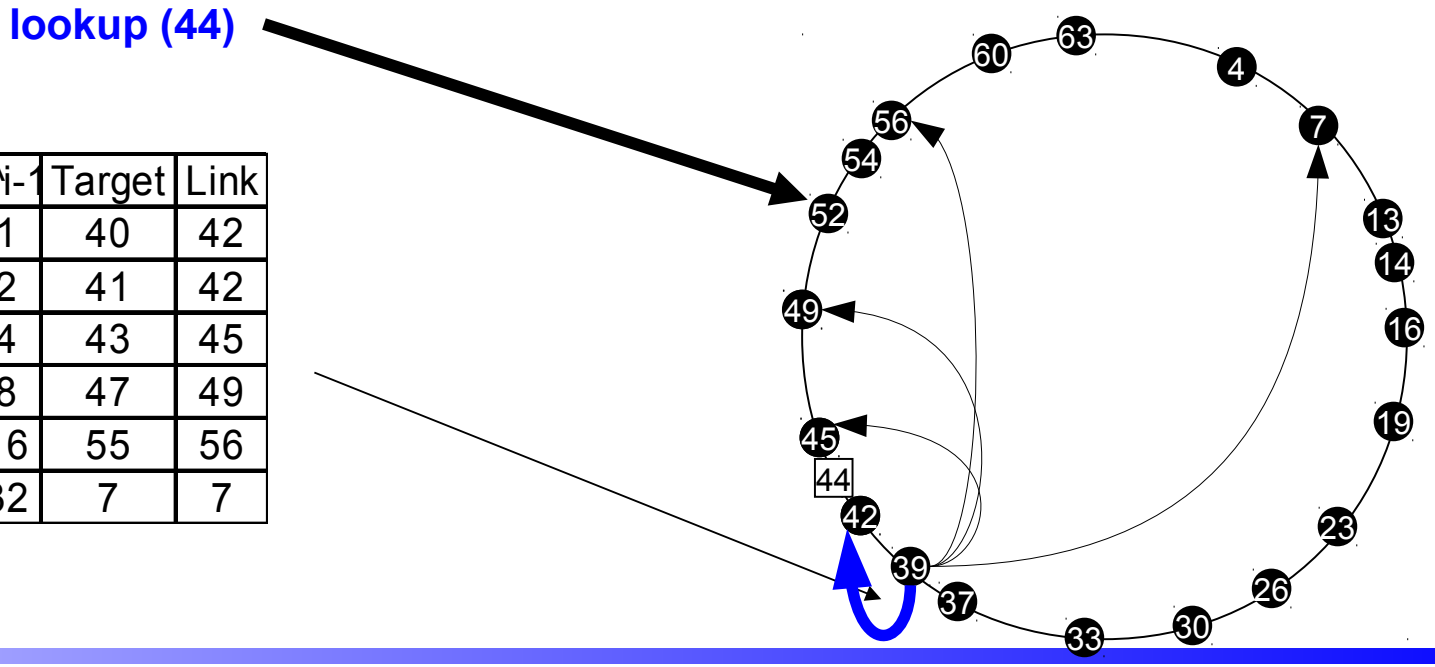
n = predecessore(k) e successore(n) = successore(k)

in questo caso il successore di n possiede la chiave



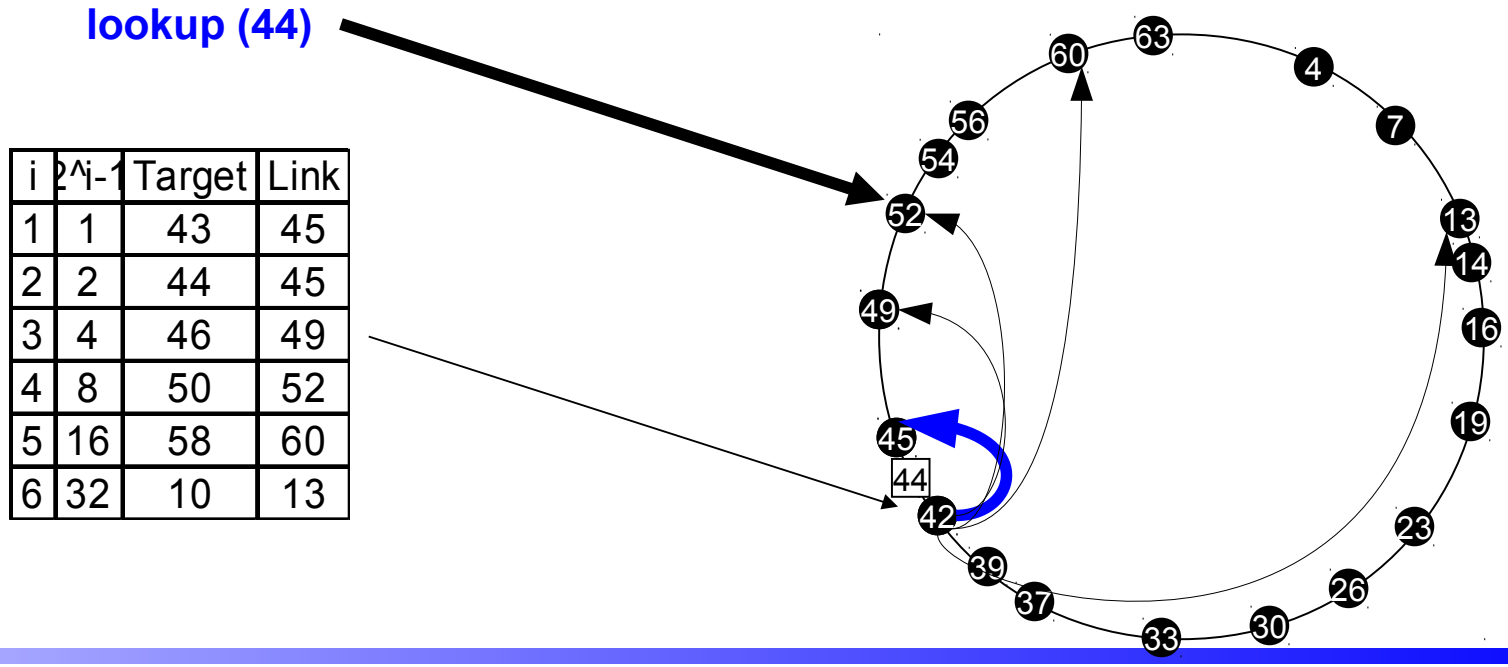
CHORD: IL ROUTING

- Algoritmo di Routing di Chord: ogni nodo n propaga la query per la chiave k al finger più distante che precede k, in senso orario
- La propagazione continua fino al nodo n tale che n = predecessore(k) e successore(n) = successore(k) in questo caso il successore di n possiede la chiave



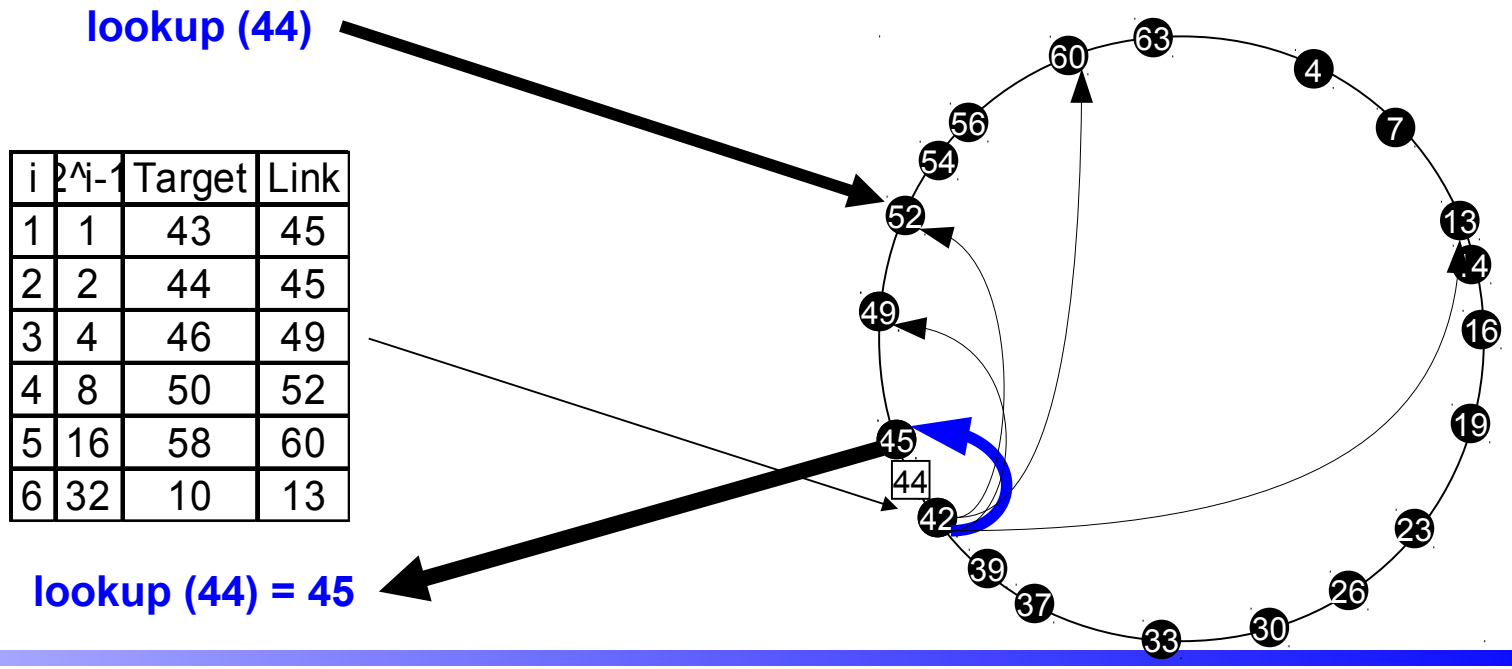
CHORD: IL ROUTING

- Algoritmo di Routing di Chord: ogni nodo n propaga la query per la chiave k al finger più distante che precede k, in senso orario
- La propagazione continua fino al nodo n tale che n = predecessore(k) e successore(n) = successore(k) in questo caso il successore di n possiede la chiave



CHORD: IL ROUTING

- Algoritmo di Routing di Chord: ogni nodo n propaga la query per la chiave k al finger più distante che precede k, in senso orario
- La propagazione continua fino al nodo n tale che n = predecessore(k) e successore(n) = successore(k) in questo caso il successore di n possiede la chiave



CHORD: L'ALGORITMO DI ROUTING

```
n.find-successor(key)
   if (key ∈ (n, successor(n)])
        return successor(n)
   else
        n' = closest_preceding_node(key);
        return n'.find-successor(key)

n.closest_preceding_node(key)
    for i = m downto 1
        if finger[i] ∈ (n,key)
            return finger[i]
```

- **n** = nodo su cui viene invocato la funzione (invocazione di metodo remoto)
- **key** = chiave ricercata
- **successor** = nodo successore di n sull'anello

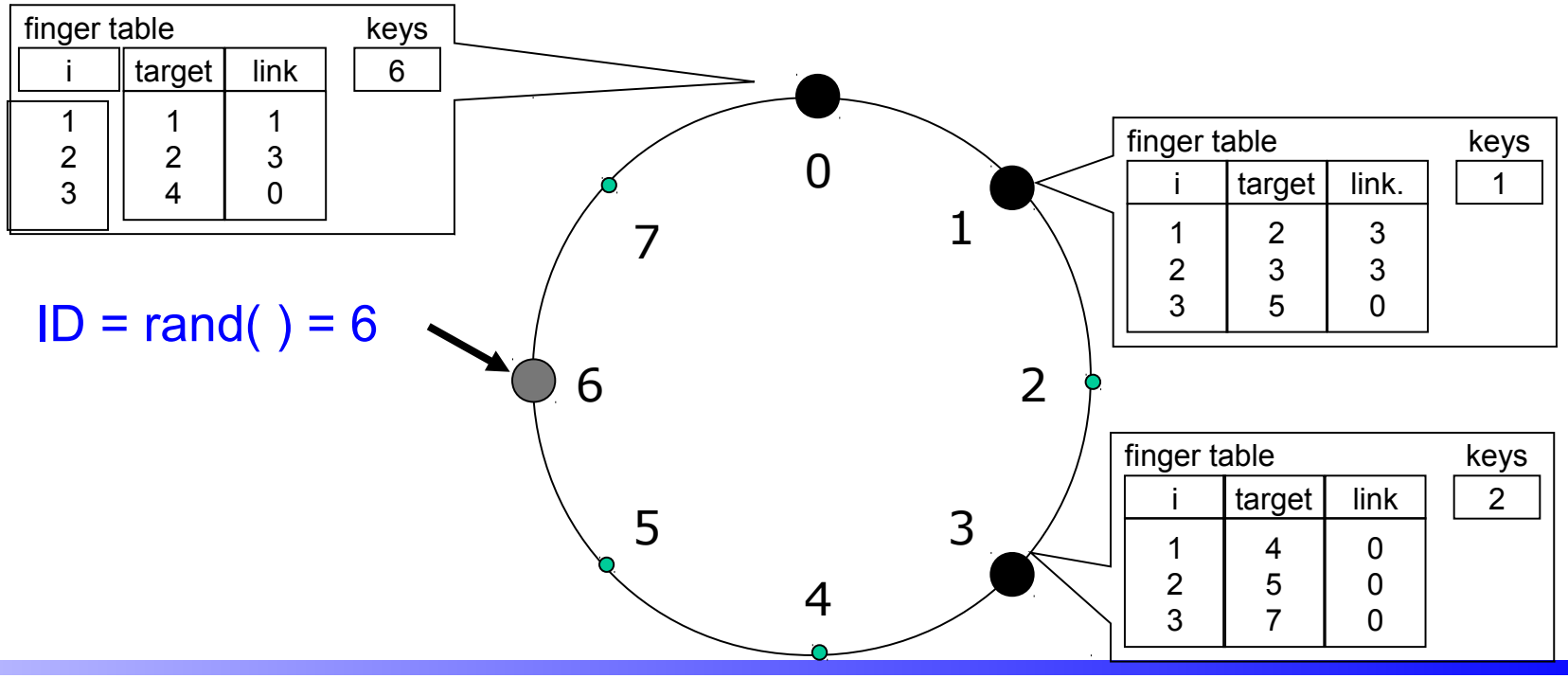
CHORD: AUTO ORGANIZZAZIONE

- Chord è in grado di gestire in modo dinamico i cambiamenti della rete
  - fallimento di nodi
  - fallimenti nella rete
  - arrivo di nuovi nodi
  - ritiro volontario dei nodi dalla rete

- Problema: mantenere consistente lo stato del sistema in presenza di cambiamenti dinamici
  - Aggiornare l'informazione necessaria per il routing
    - Correttezza del Routing: è necessario che ogni nodo mantenga aggiornato il suo successore effettivo sull'anello
    - Efficienza del Routing: dipende dall'aggiornamento tempestivo delle finger tables
  - Tolleranza ai guasti

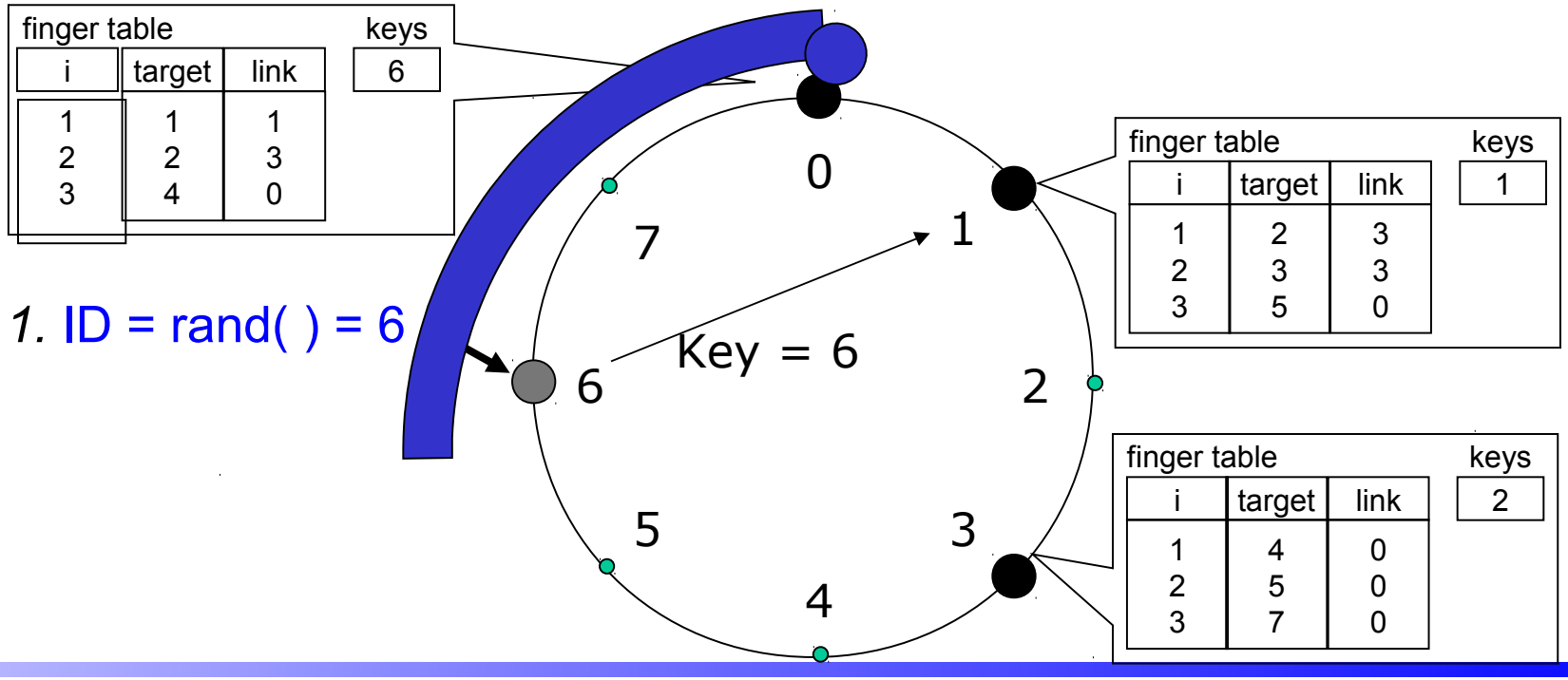
INSERIMENTO DI NODI: PASSO 1

1. il nuovo nodo n sceglie un identificatore ID
2. n contatta il bootstrap node), si aggancia al suo successore
3. costruisce la propria finger table (può essere lazy)
4. riceve le chiavi che deve gestire
5. stabilizzazione dei nodi successori/predecessori
6. stabilizzazione finger table di tutti i nodi



INSERIMENTO DI NODI: PASSO 2

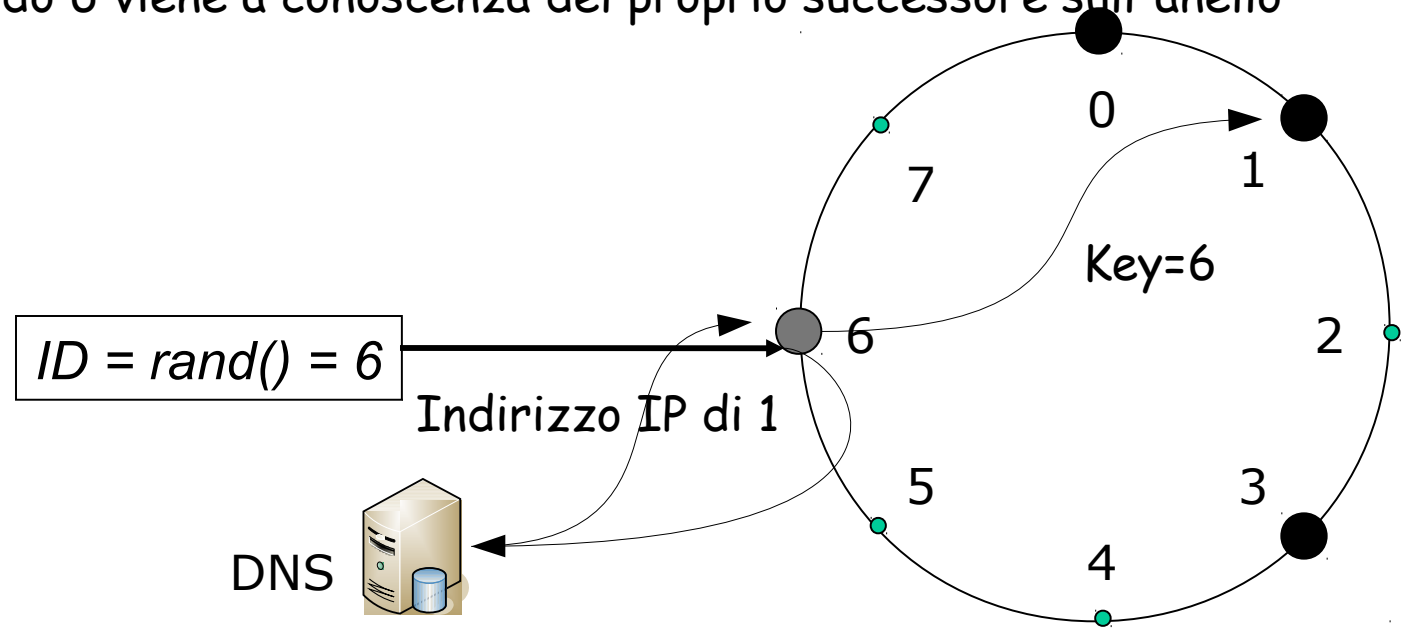
1. il nuovo nodo n sceglie un identificatore ID
2. n contatta il bootstrap node, si aggancia al suo successore
3. costruisce la propria finger table (può essere lazy)
4. riceve le chiavi che deve gestire
5. stabilizzazione dei nodi successori/predecessori
6. stabilizzazione finger table di tutti i nodi



INSERIMENTO DI NUOVI NODI: PASSO 3

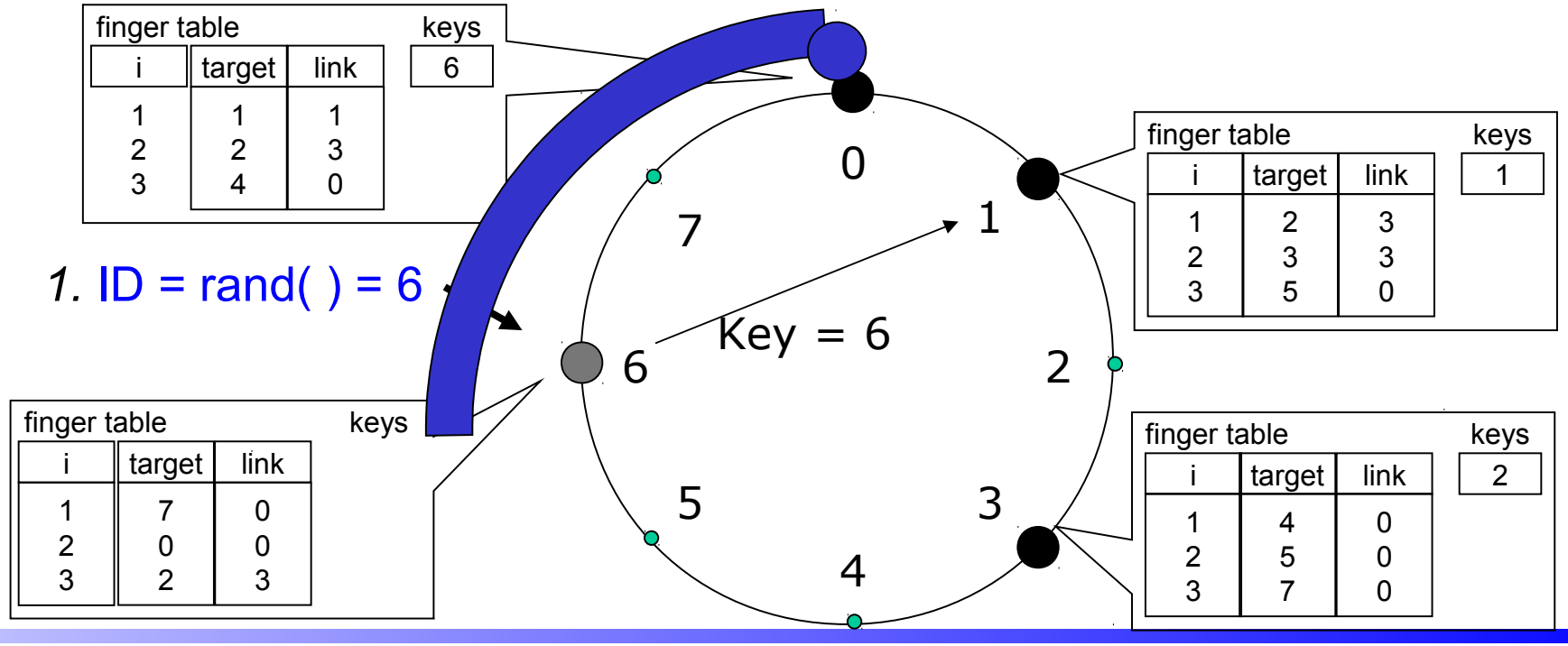
Individuazione del nodo successore sull'anello

- Ricorda: il routing di una chiave individua il successore della chiave
- il nuovo nodo (6) invia al nodo di bootstrap (1) una query con chiave uguale al proprio identificatore(6)
  - il routing standard di Chord individua il successore della chiave sull'anello
  - nel nostro caso, il successore del nodo 6, cioè il nodo 0
- in questo modo 6 viene a conoscenza del proprio successore sull'anello



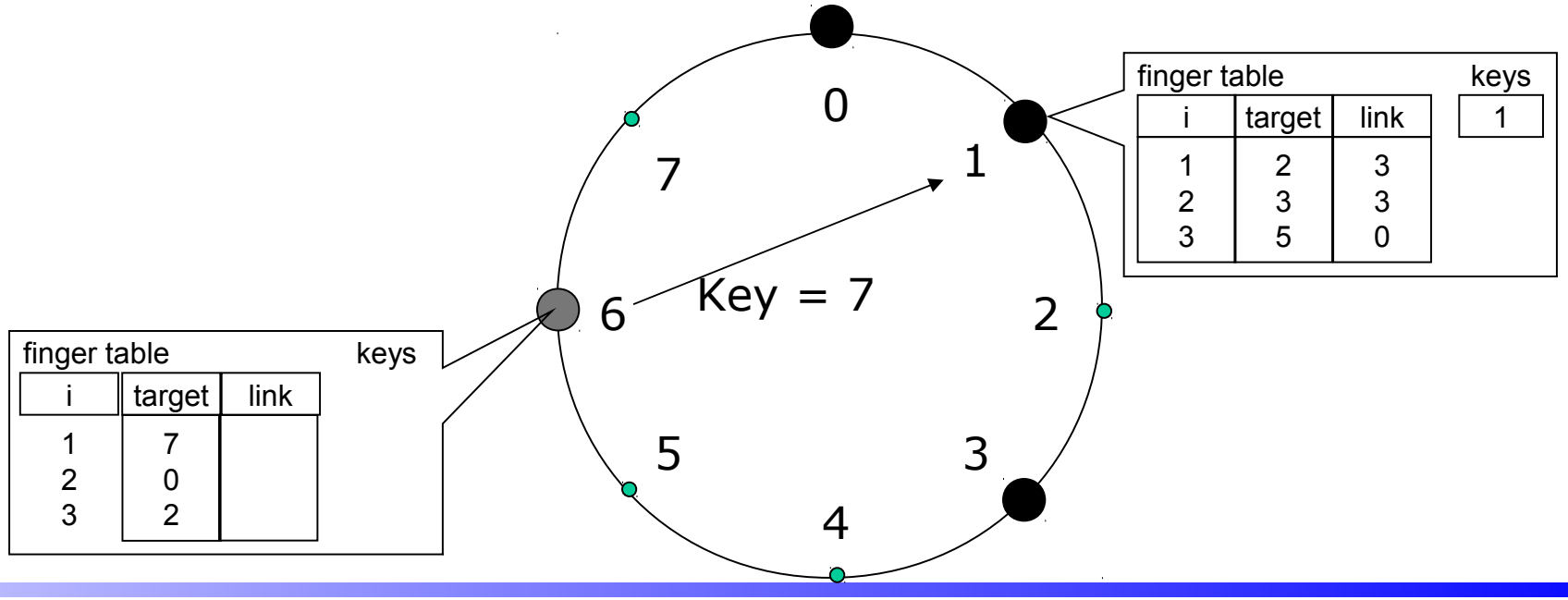
INSERIMENTO DI NODI: PASSO 3

1. il nuovo nodo n sceglie un identificatore ID
2. n contatta il bootstrap node, si aggancia al suo successore
3. costruisce la propria finger table (può essere lazy)
4. riceve le chiavi che deve gestire
5. stabilizzazione dei nodi successori/predecessori
6. stabilizzazione finger table di tutti i nodi



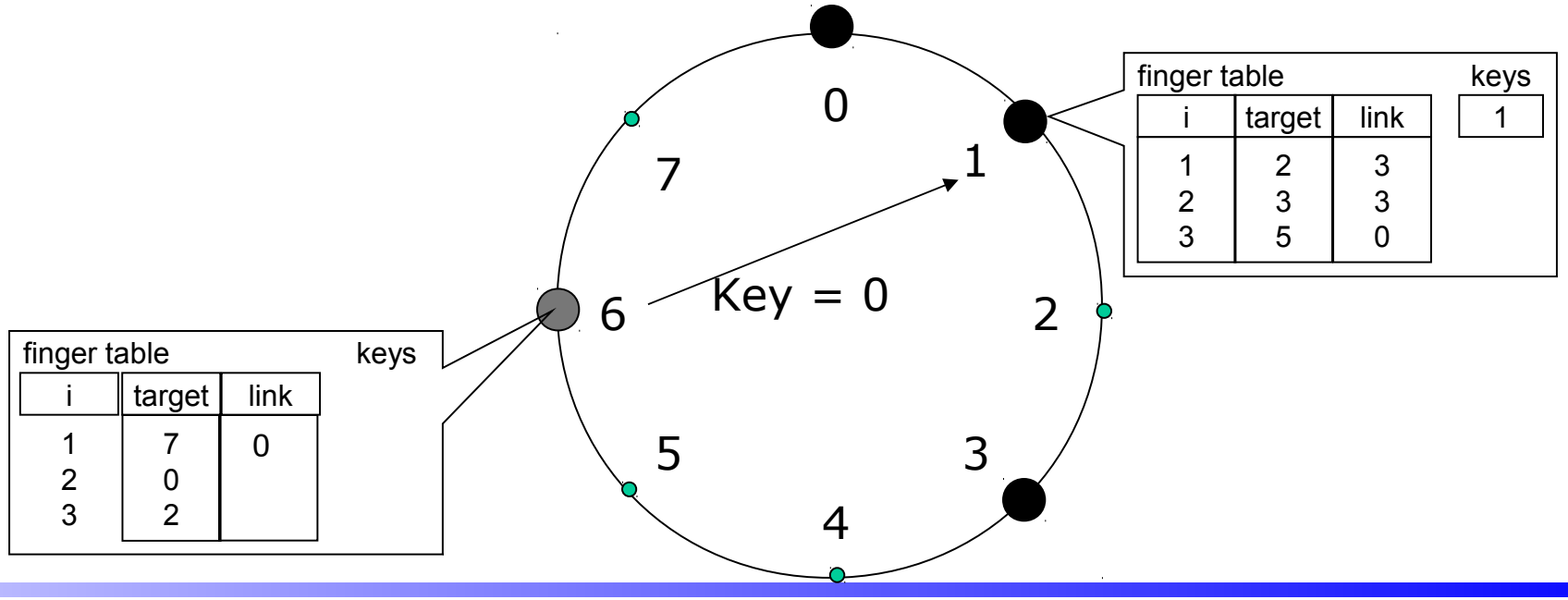
COSTRUZIONE DELLA FINGER TABLE

- definire una struttura di m entrate
- iterare sulle righe della finger table ed individuare il target per quella riga (ricorda finger[i] punta a successor($n + 2^{i-1}$), $1 \leq i \leq m$)
- per ogni riga: inviare una query al nodo di bootstrap per trovare il successore del target: questo è il link che deve essere inserito nella finger table



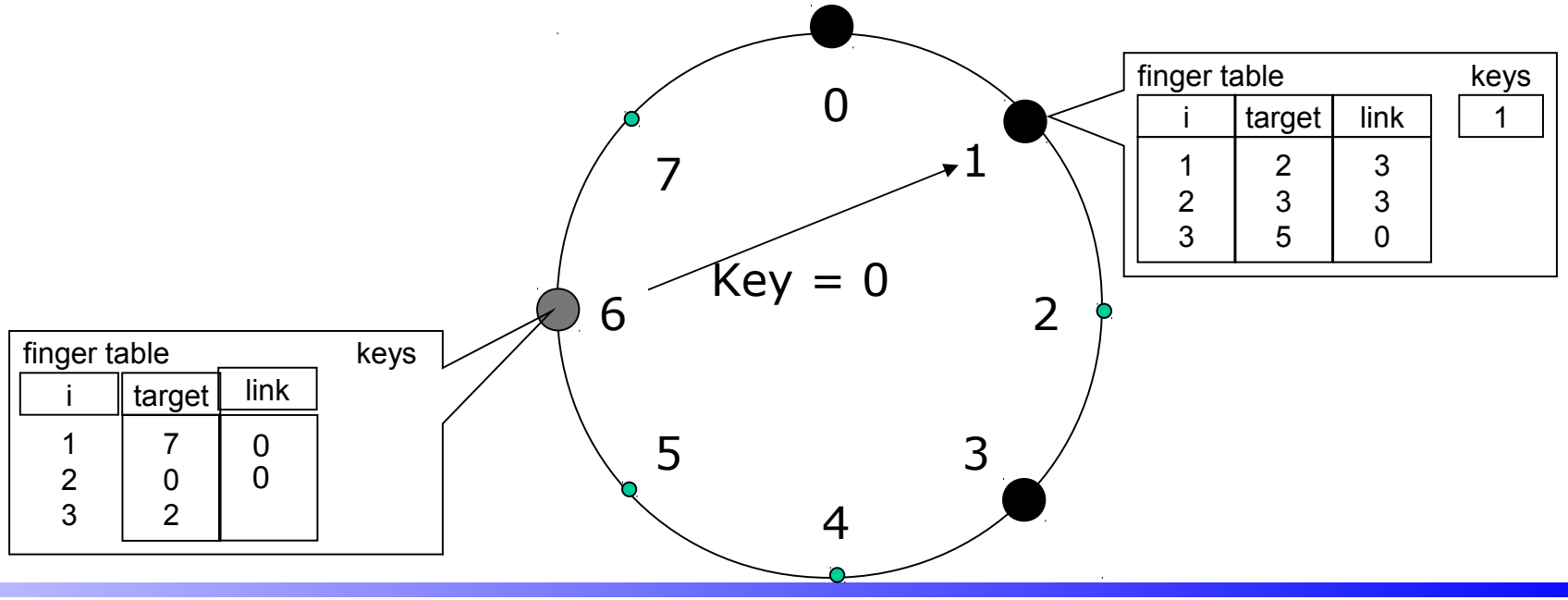
COSTRUZIONE DELLA FINGER TABLE

- definire una struttura di m entrate
- iterare sulle righe della finger table ed individuare il target per quella riga (ricorda finger[i] punta a successor($n + 2^{i-1}$), $1 \leq i \leq m$)
- per ogni riga: inviare una query al nodo di bootstrap per trovare il successore del target: questo è il link che deve essere inserito nella finger table



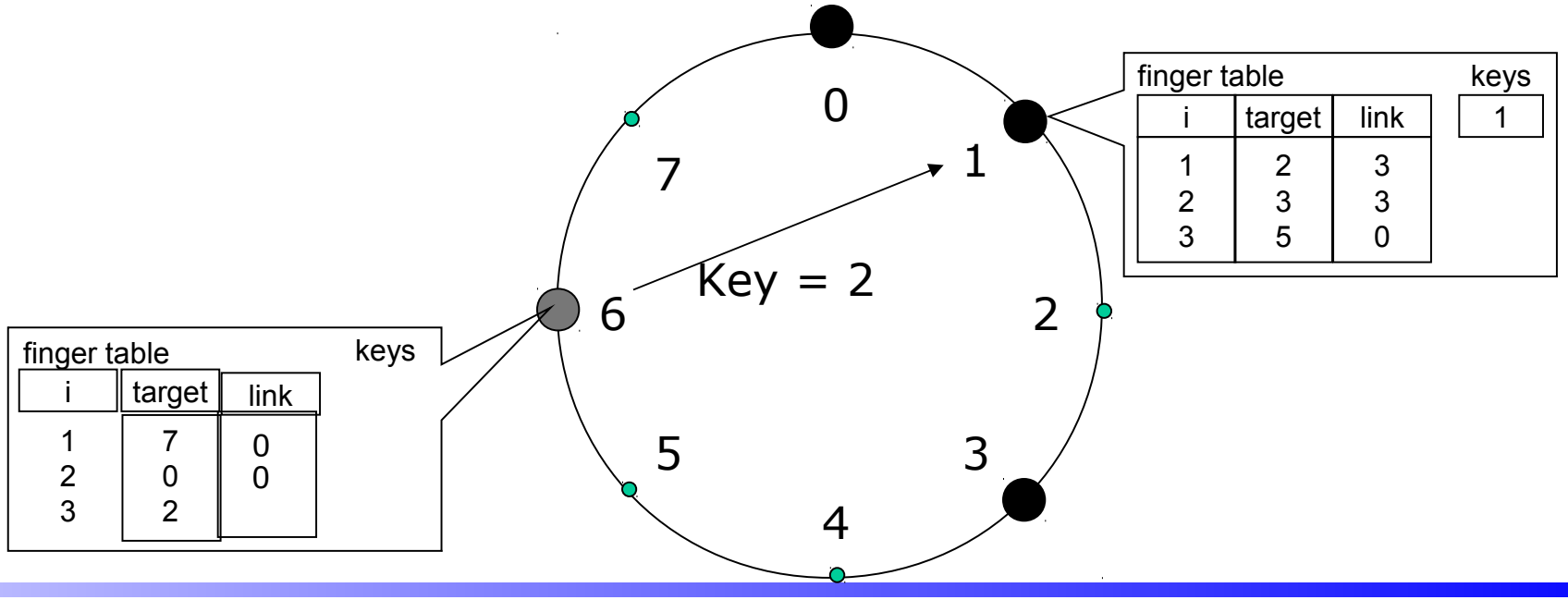
COSTRUZIONE DELLA FINGER TABLE

- definire una struttura di m entrate
- iterare sulle righe della finger table ed individuare il target per quella riga (ricorda finger[i] punta a successor($n + 2^{i-1}$), $1 \leq i \leq m$)
- per ogni riga: inviare una query al nodo di bootstrap per trovare il successore del target: questo è il link che deve essere inserito nella finger table



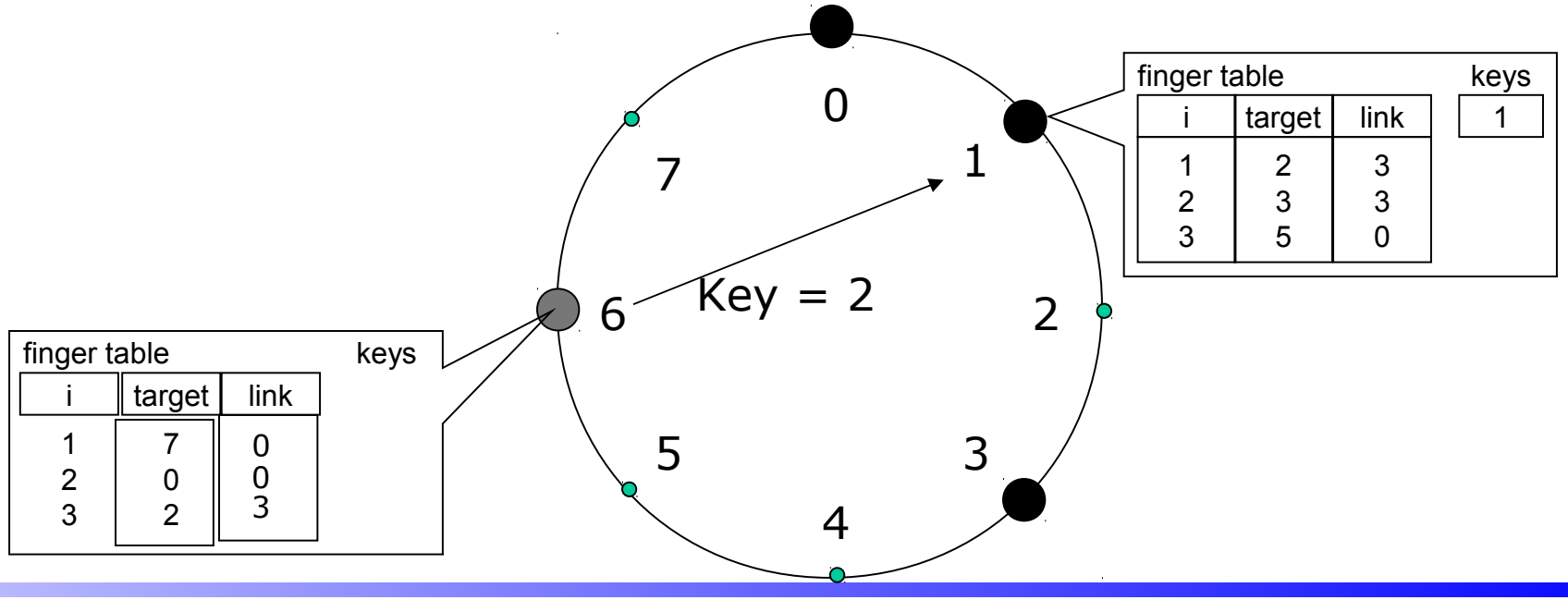
COSTRUZIONE DELLA FINGER TABLE

- definire una struttura di m entrate
- iterare sulle righe della finger table ed individuare il target per quella riga (ricorda finger[i] punta a successor($n + 2^{i-1}$), $1 \leq i \leq m$)
- per ogni riga: inviare una query al nodo di bootstrap per trovare il successore del target: questo è il link che deve essere inserito nella finger table



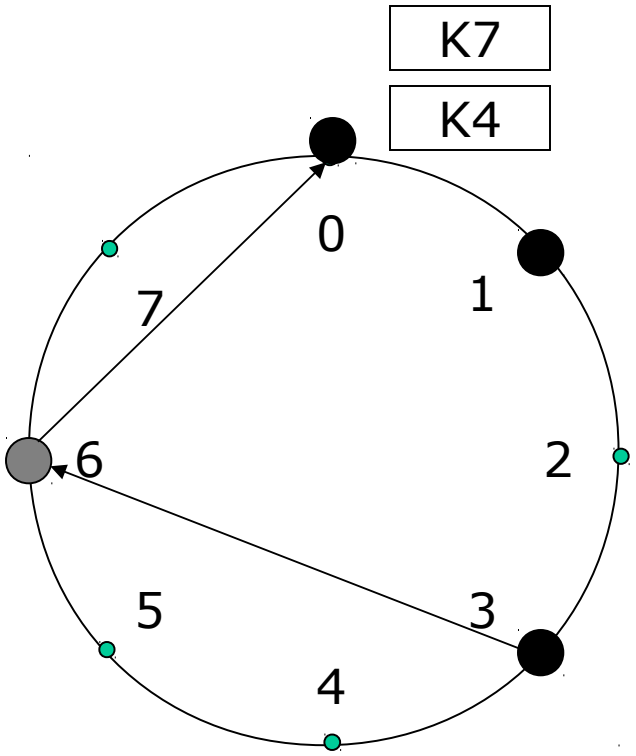
COSTRUZIONE DELLA FINGER TABLE

- definire una struttura di m entrate
- iterare sulle righe della finger table ed individuare il target per quella riga (ricorda finger[i] punta a successor($n + 2^{i-1}$), $1 \leq i \leq m$)
- per ogni riga: inviare una query al nodo di bootstrap per trovare il successore del target: questo è il link che deve essere inserito nella finger table



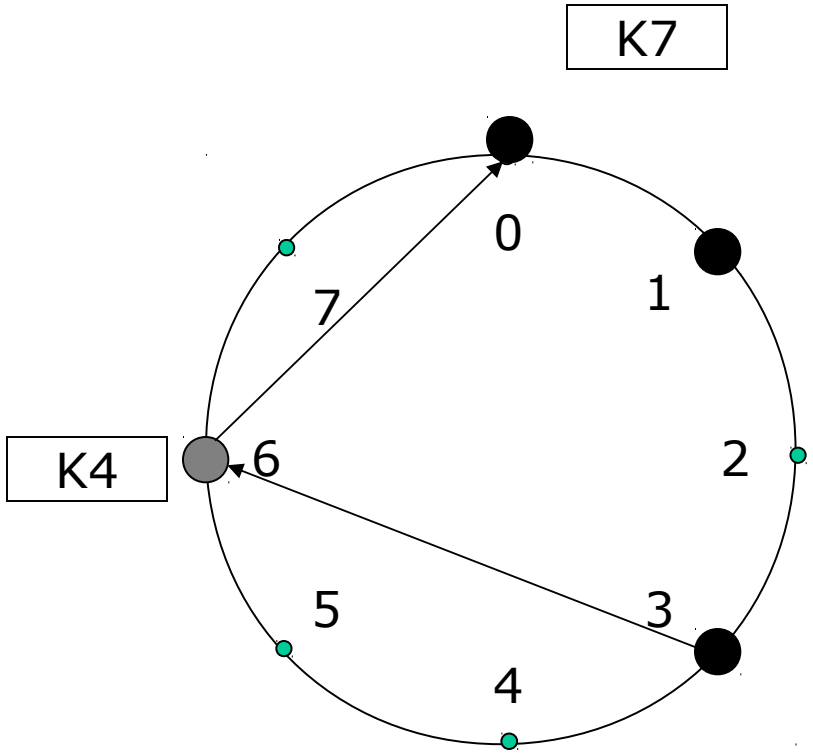
RICEZIONE CHIAVI DA GESTIRE

- Per rispettare la strategia di allocazione chiavi-nodi,il nodo 6 deve copiare tutte le chiavi minori o uguali a 6 dal nodo 0. Questo deve essere effettuato dalla applicazione



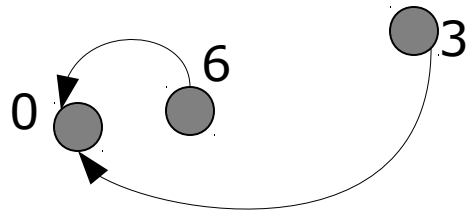
RICEZIONE CHIAVI DA GESTIRE

- Per rispettare la strategia di allocazione chiavi-nodi,il nodo 6 deve copiare tutte le chiavi minori o uguali a 6 dal nodo 0. Questo deve essere effettuato dalla applicazione



INSERIMENTO DI NODI: STABILIZZAZIONE

- dopo i passi illustrati in precedenza, il nuovo nodo n conosce il suo successore sull'anello e possiede una finger table valida, ma gli altri nodi dell'anello non sono consapevoli della presenza di n. Con riferimento all'esempio precedente, il nodo 3 vede ancora il nodo 0 come proprio successore



- I nodi eseguono periodicamente alcune procedure di stabilizzazione per acquisire conoscenza sui nuovi nodi che si sono inseriti nella rete

- stabilize ( ) aggiorna il puntatore al successore

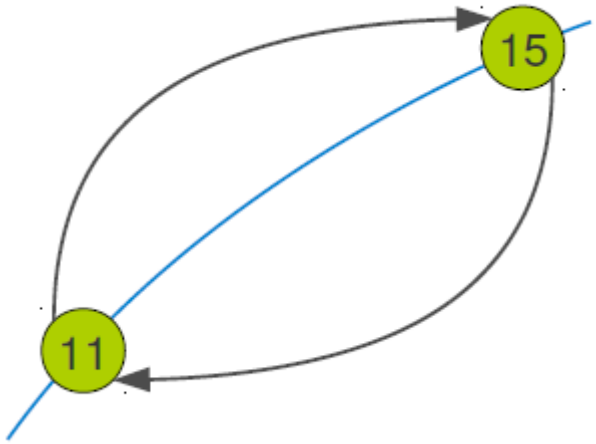
- fix fingers ( ) aggiorna la finger table di ogni nodo, inserendovi eventuali riferimenti a nuovi nodi

CHORD: STABILIZE

1. il nuovo nodo n sceglie un identificatore ID
2. n contatta il bootstrap node, si aggancia al suo successore
3. costruisce la propria finger table (può essere lazy)
4. riceve le chiavi che deve gestire
5. stabilizzazione dei nodi successori/predecessori
6. stabilizzazione finger table di tutti i nodi

```
// Periodically at n:
v := succ.pred
if (v ≠ nil and v ∈ (n,succ]) then
    set succ := v
send a notify(n) to succ
```

```
// When receiving notify(p) at n:
if (pred = nil or p ∈ (pred, n]) then
    set pred := p
```

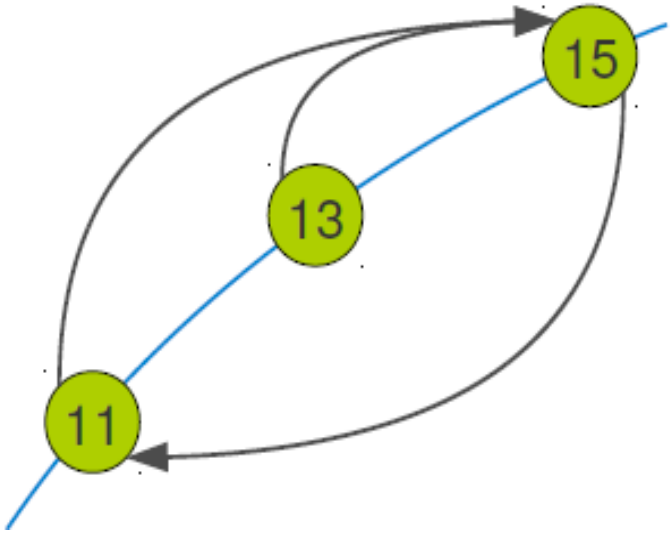


CHORD: STABILIZE

1. il nuovo nodo n sceglie un identificatore ID
2. n contatta il bootstrap node, si aggancia al suo successore
3. costruisce la propria finger table (può essere lazy)
4. riceve le chiavi che deve gestire
5. stabilizzazione dei nodi successori/predecessori
6. stabilizzazione finger table di tutti i nodi

```
// Periodically at n:
v := succ.pred
if (v ≠ nil and v ∈ (n,succ]) then
    set succ := v
send a notify(n) to succ
```

```
// When receiving notify(p) at n:
if (pred = nil or p ∈ (pred, n]) then
    set pred := p
```

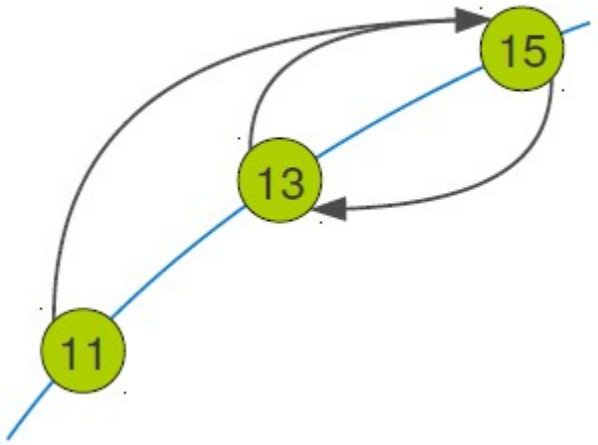


CHORD: STABILIZE

1. il nuovo nodo n sceglie un identificatore ID
2. n contatta il bootstrap node, si aggancia al suo successore
3. costruisce la propria finger table (può essere lazy)
4. riceve le chiavi che deve gestire
5. stabilizzazione dei nodi successori/predecessori
6. stabilizzazione finger table di tutti i nodi

```
// Periodically at n:
v := succ.pred
if (v ≠ nil and v ∈ (n,succ]) then
    set succ := v
send a notify(n) to succ
```

```
// When receiving notify(p) at n:
if (pred = nil or p ∈ (pred, n]) then
    set pred := p
```

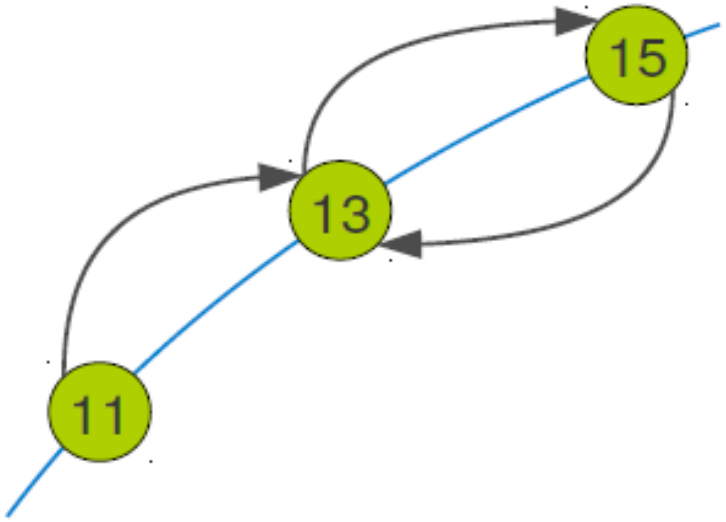


CHORD: STABILIZE

1. il nuovo nodo n sceglie un identificatore ID
2. n contatta il bootstrap node, si aggancia al suo successore
3. costruisce la propria finger table (può essere lazy)
4. riceve le chiavi che deve gestire
5. stabilizzazione dei nodi successori/predecessori
6. stabilizzazione finger table di tutti i nodi

```
// Periodically at n:
v := succ.pred
if (v ≠ nil and v ∈ (n,succ]) then
    set succ := v
send a notify(n) to succ
```

```
// When receiving notify(p) at n:
if (pred = nil or p ∈ (pred, n]) then
    set pred := p
```

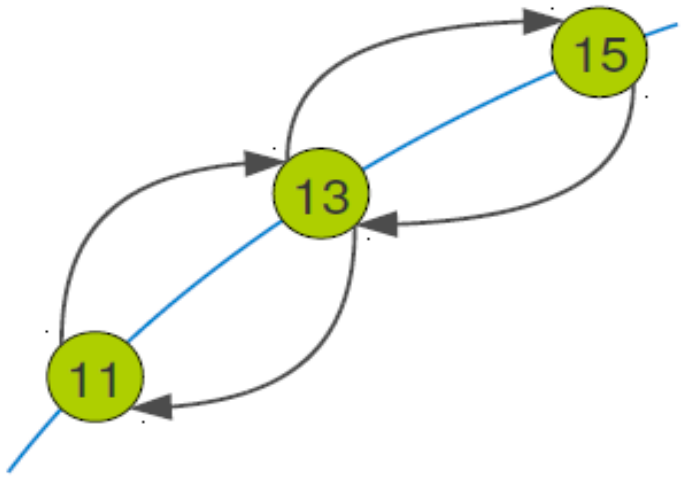


CHORD: STABILIZE

1. il nuovo nodo n sceglie un identificatore ID
2. n contatta il bootstrap node, si aggancia al suo successore
3. costruisce la propria finger table (può essere lazy)
4. riceve le chiavi che deve gestire
5. stabilizzazione dei nodi successori/predecessori
6. stabilizzazione finger table di tutti i nodi

```
// Periodically at n:
v := succ.pred
if (v ≠ nil and v ∈ (n,succ]) then
    set succ := v
send a notify(n) to succ
```

```
// When receiving notify(p) at n:
if (pred = nil or p ∈ (pred, n]) then
    set pred := p
```



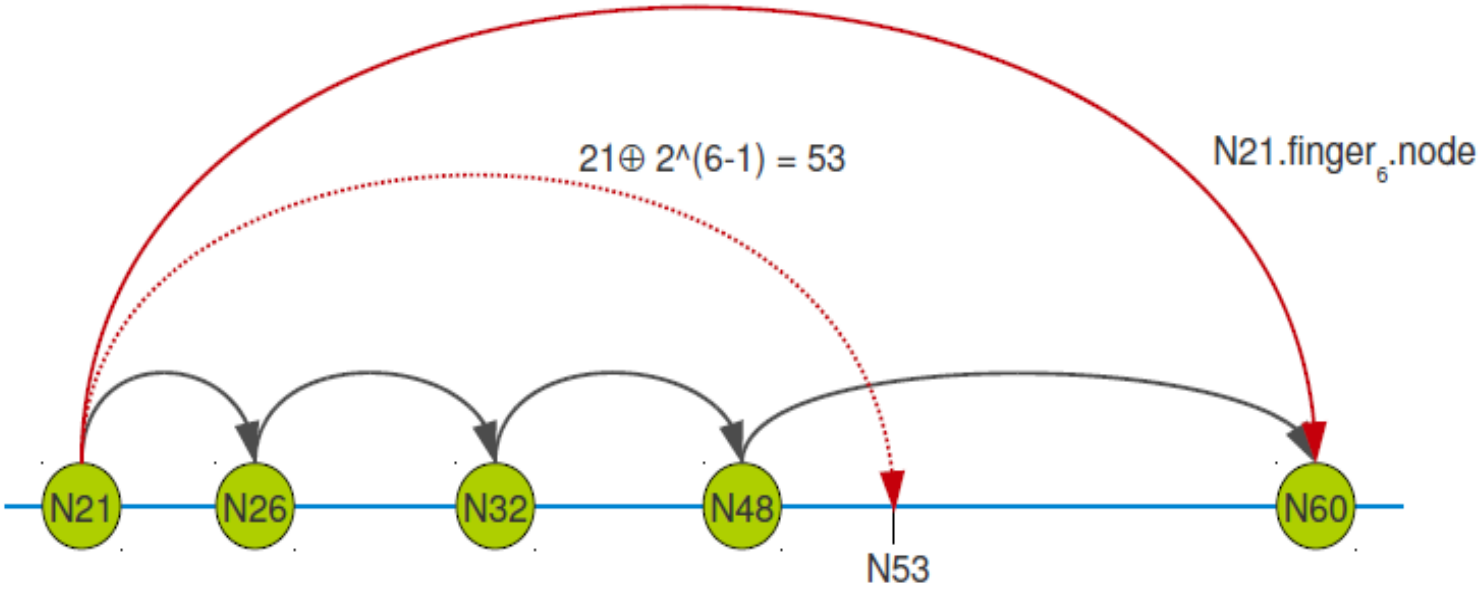
PAASO 6: FIX FINGERS

- Si effettua periodicamente un refresh delle finger table e si memorizzal'indice del prossimo finger da fixare
- La variabile next è impostata inizialmente a 0

```
// When receiving notify(p) at n:
procedure n.fixFingers() {
   next := next+1
   if (next > m) then
      next := 1
   finger[next] := findSuccessor(n ⊕ 2^(next - 1))
}
```

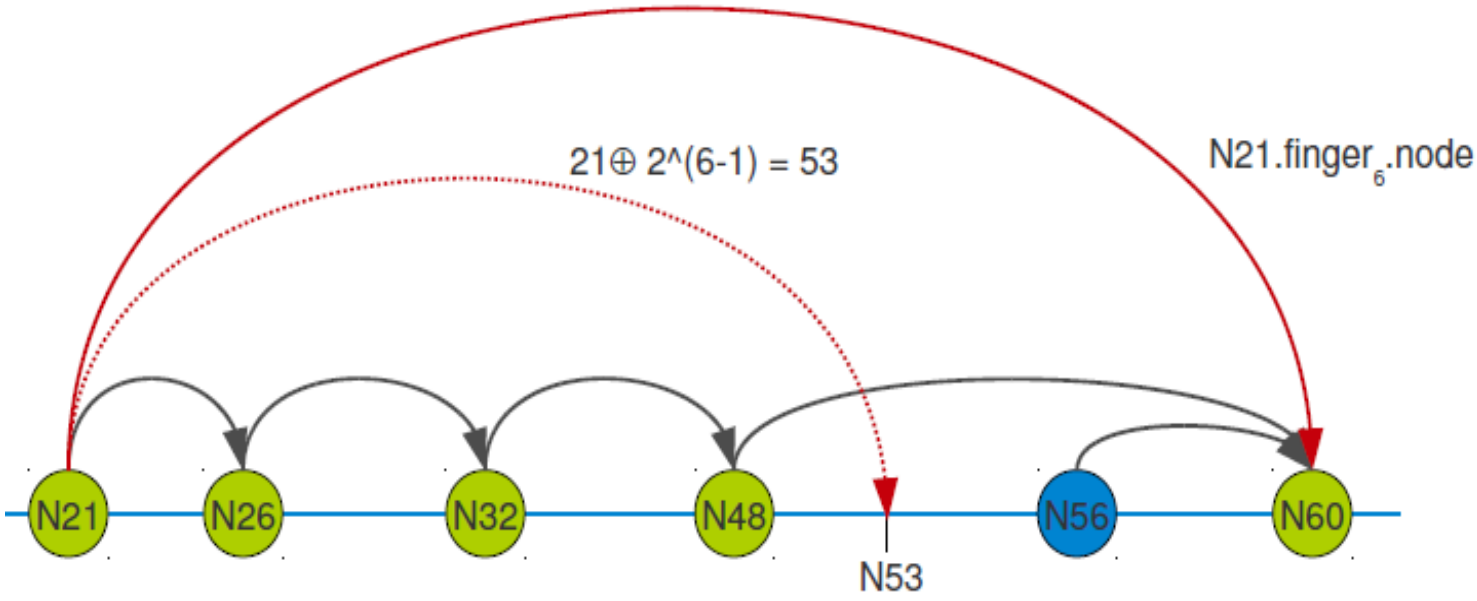
PASSO 6: FIX FINGERS

- Situazione corrente succ(N48) = N60
- Finger 6 di N21: Succ(21+ 2^(6-1)) = Succ(53) = N60.



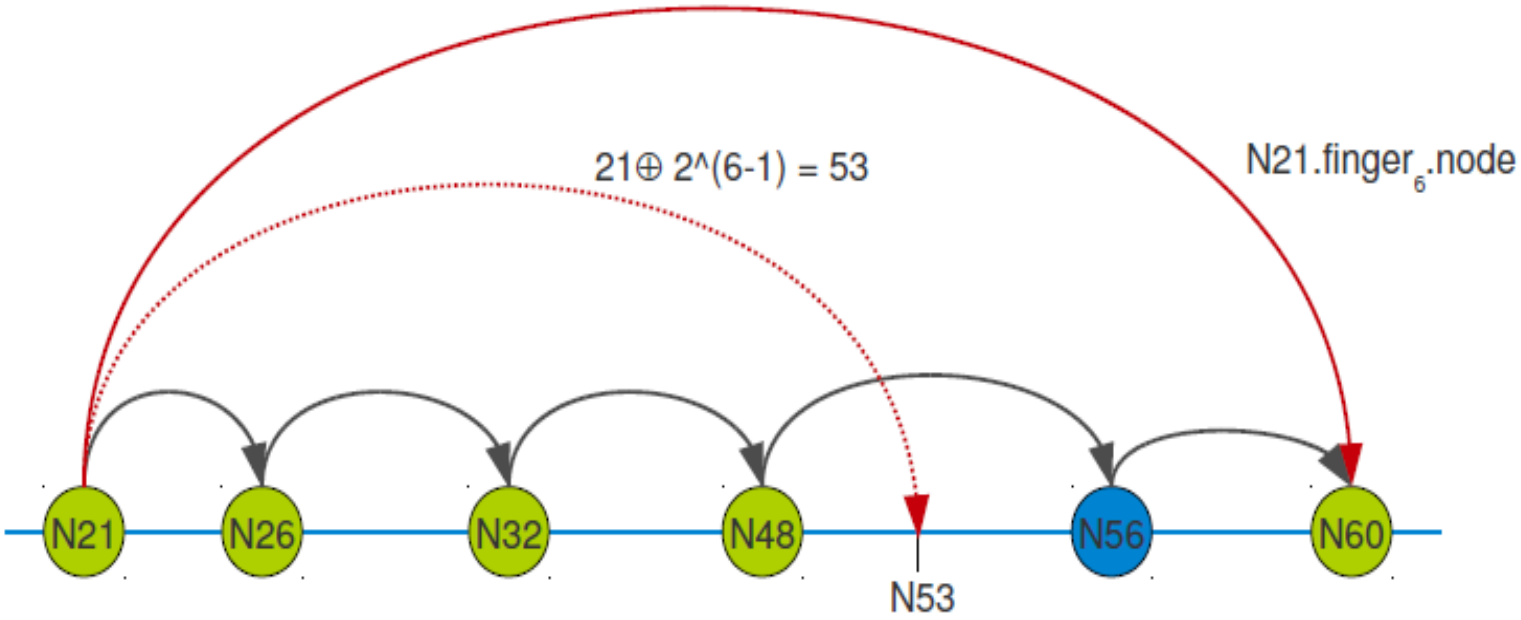
PASSO 6: FIX FINGERS

- Succ(21+ 2^(6-1)) = Succ(53) = ?
- Il nuovo nodo N56 entra nella rete e si collega con il suo successore
- Il Finger 6 del nodo N21 risulta ora errato
- N21 tenta di “fixare” il suo finger 6 effettuando un look-up di 53. Tuttavia N48 non ha ancora fixato il suo successore, per cui il finger 6 di N21 rimane errato



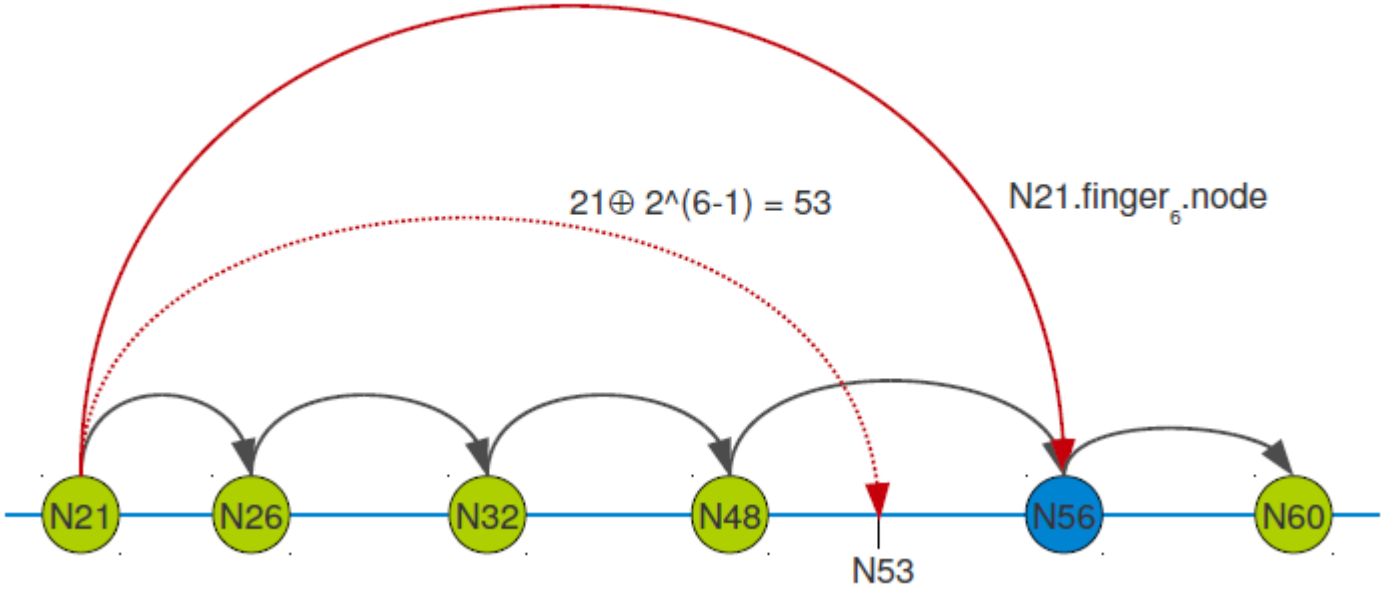
PASSO 6: FIX FINGERS

- Succ(21+ 2^(6-1)) = Succ(53) = ?
- N48 stabilizza il suo successore
- L'anello risulta corretto



PASSO 6: FIX FINGERS

- Succ(21+ 2^(6-1))= Succ(53) = N56
- Al successivo tentativo da parte di N21 di fissare il suo Finger 6, la risposta di N48 ora e' corretta ed N21 puo' correggere il proprio finger



CHORD: INSERIMENTO DI NODI

- Lazy Join:
  - Quando un nodo si inserisce sull'anello può inizializzare solo il suo successore
  - Periodicamente rinfresca il contenuto della propria finger table (fix.finger)
- La correttezza della ricerca dipende dalla corretta impostazione del nodo successore di ogni nodo appartenente all'anello Chord
- La migrazione delle risorse al nuovo nodo non è gestita da Chord, deve essere eseguita dalla applicazione
- Tutti i link di un nodo sono correttamente aggiornati solo quando
  - è stata eseguita la stabilize( ) (che aggiorna il successore di un nodo considerando eventuali nodi inseriti)
  - è stata eseguita la fixfingers( ) per tutti i finger della finger table sia del nodo che ha effettuat la join() sia degli altri nodi

CHORD: INSERIMENTO DI NODI

Esito di una ricerca eseguita prima che il sistema sia ritornato in uno stato stabile, dopo un inserimento:

- non tutti i puntatori ai nodi successivi sono corretti oppure le chiavi non sono state completamente trasferite. La ricerca può fallire ed occorre ritentare in seguito

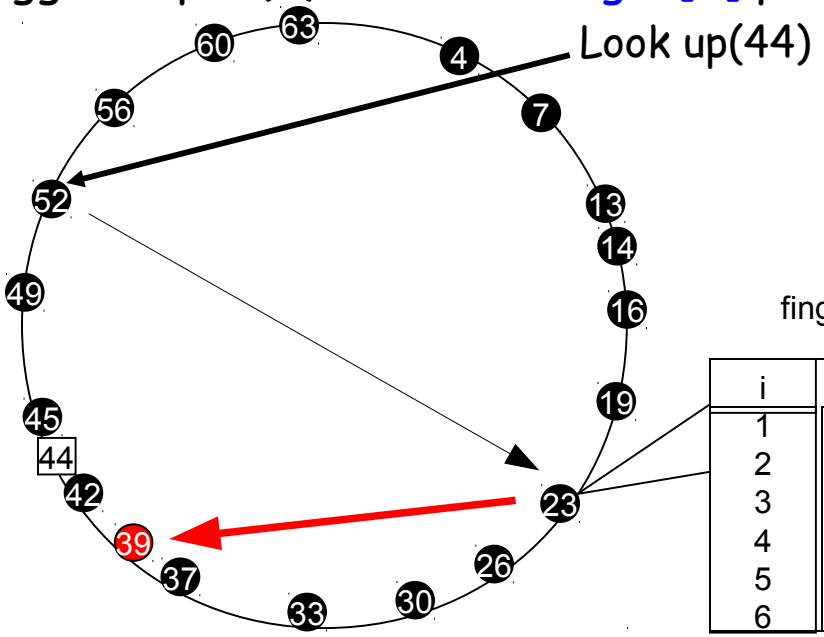
- ogni nodo ha aggiornato il suo successore reale sull'anello e le chiavi sono state correttamente trasferite tra nodi, le fingers table possono contenere informazione non completamente aggiornata. La ricerca ha esito positivo , ma può essere rallentata

- tutte le finger tables sono in uno stato "raginevolmente aggiornato", allora il routing viene garantito in $O(\log N)$ passi

FALLIMENTO DI NODI: CASO 1

ogni comunicazione con i fingers è controllata mediante time outs. Se il time-out scade :

- la query viene inviata al finger precedente, per evitare di oltrepassare il nodo destinazione.
- il finger caduto viene rimpiazzato con il suo successore nella finger table (trigger repair) (ricordare finger[i] punta a successor($n + 2^{i-1}$), $1 \leq i \leq m$)



finger table

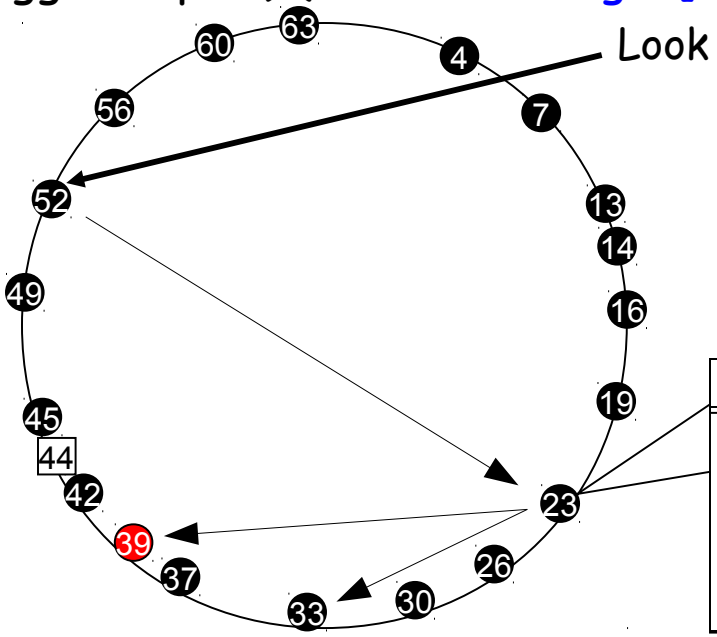
i	target	link
1	24	26
2	25	26
3	27	30
4	31	33
5	39	39
6	54	56

Esempio: fallisce il nodo 39
il nodo 23 deve 'riparare' la propria finger table che contiene un puntatore al 39

FALLIMENTO DI NODI: CASO 1

durante il routing, ogni comunicazione con i fingers è controllata mediante l'attivazione di time outs. Se il time-out scade :

- la query viene inviata al finger precedente, per evitare di oltrepassare il nodo destinazione.
- il finger caduto viene rimpiazzato con il suo successore nella finger table (trigger repair) (ricordare finger[i] punta a successor($n + 2^{i-1}$), $1 \leq i \leq m$)



finger table

i	target	link
1	24	26
2	25	26
3	27	30
4	31	33
5	39	56
6	54	56

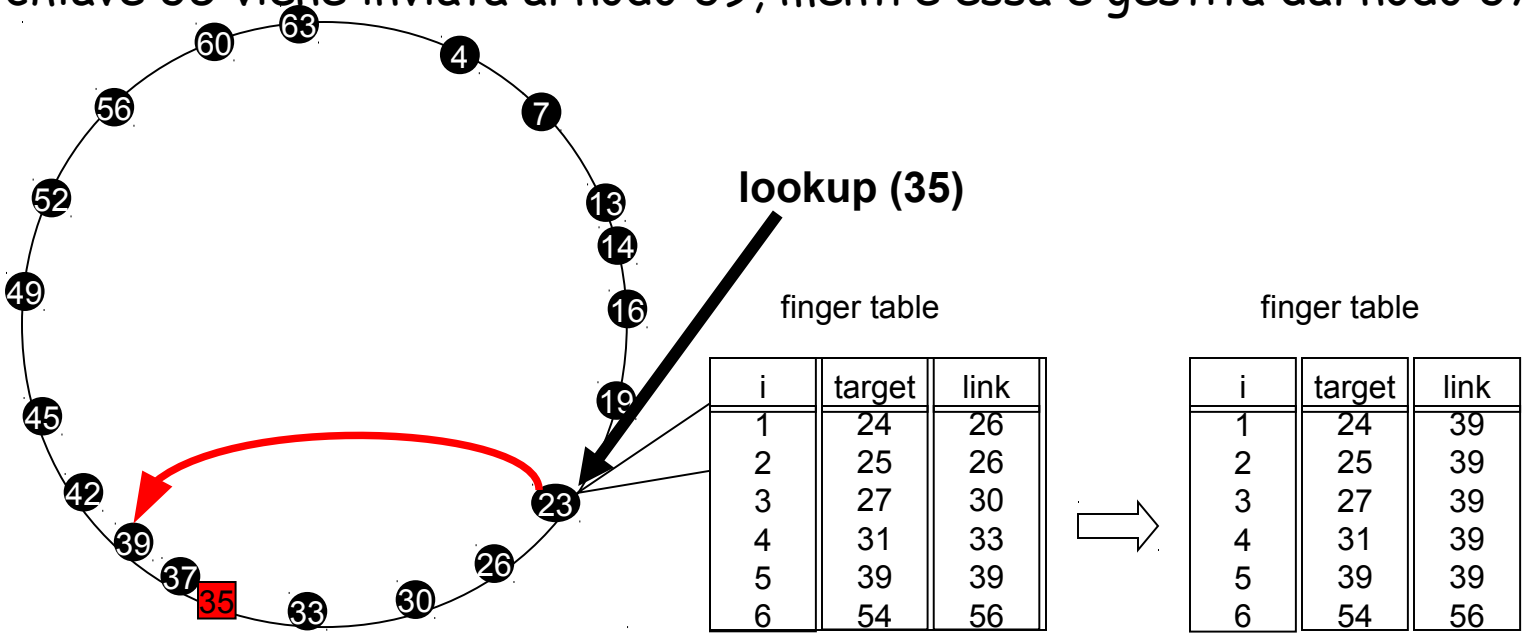
Esempio:
Il nodo 23 sostituisce nella finger table il link successivo a 39

La chiave 44 viene inviata al 33 invece che al 39

FALLIMENTO DI NODI: CASO 2

Situazione inconsistente: un nodo perde il riferimento al suo vero successore sull'anello

- i primi tre successori del nodo 23 (26,30,33) falliscono.
- il successore sull'anello del nodo 23 diventa il nodo 37
- poichè il nodo 23 non possiede un riferimento al nodo 37 nella finger table, 23 considera 39 come suo nuovo successore.
- La chiave 35 viene inviata al nodo 39, mentre essa è gestita dal nodo 37.



FALLIMENTO DI NODI

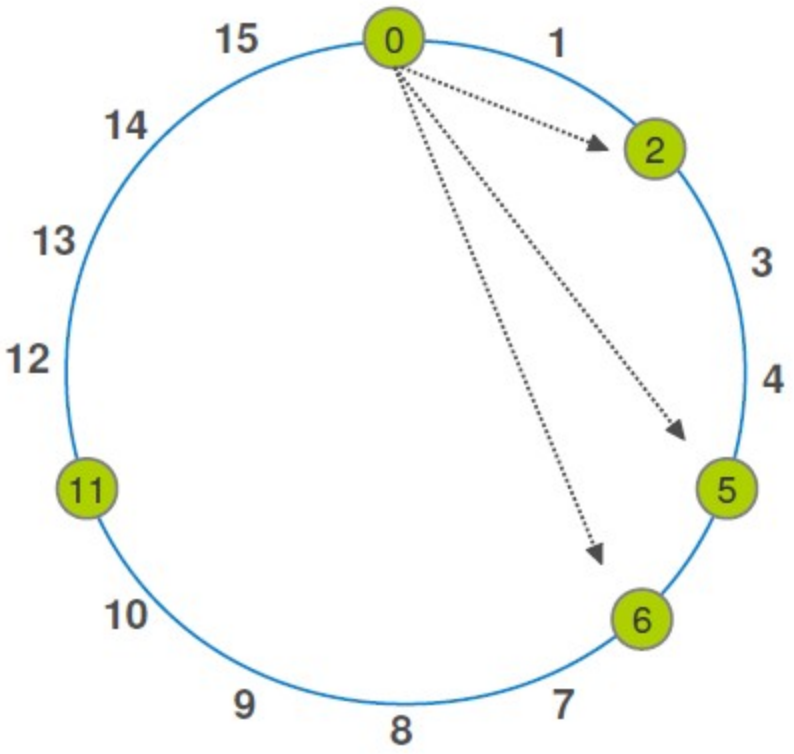
- la correttezza dell'algoritmo di routing è garantita se ogni nodo mantiene aggiornato il riferimento al nodo successivo anche in caso di fallimenti multipli

- ma… in caso di fallimenti simultanei questo invariante non può essere garantito

- Per migliorare la robustezza del sistema
  - ogni nodo n mantiene una lista dei suoi r immediati successori sull'anello (r=log N)
  - se n rileva la caduta del suo successore, il successore viene sostituito con il secondo elemento della lista e così via
  - la lista viene mantenuta consistente mediante la procedura di stabilizzazione
  - valori crescenti di r rendono il sistema maggiormente robusto

FALLIMENTO DI NODI

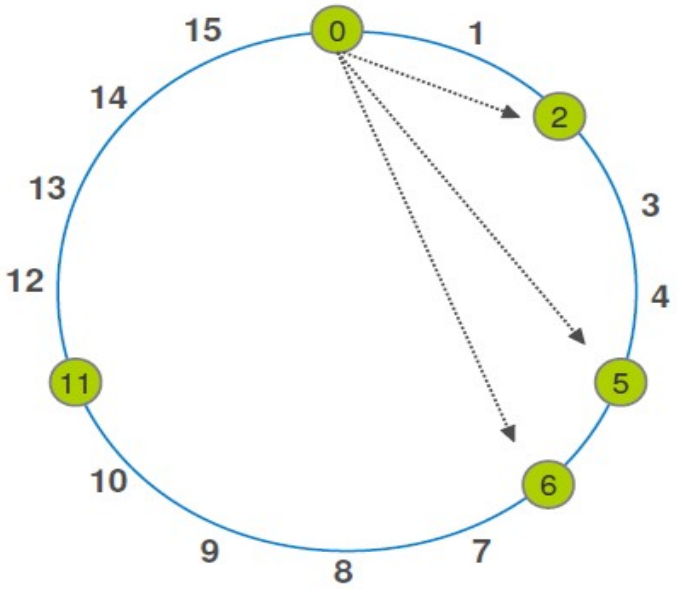
- un nodo mantiene una lista di dimensione r contenente i suoi immediati r successori
  - succ(x+1)
  - succ(succ(x+1)+1)
  - (succ(succ(x+1)+1)+1)
- r ≅ log(N), N numero totale di nodi sull'anello




```
// join a Chord ring containing node m
procedure n.join(m) {
   pred := nil
   Succ := m.findSuccessor(n)
  updateSuccesorList(succ.successorList)
}
```

FALLIMENTO DI NODI

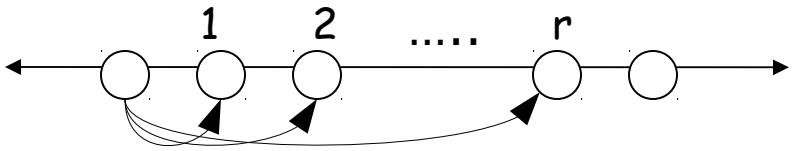
- un nodo x mantiene una lista di dimensione r contenente i suoi immediati r successori
  - succ(x+1)
  - succ(succ(x+1)+1)
  - (succ(succ(x+1)+1)+1)
- r ≅ log(N), N numero totale di nodi sull'anello

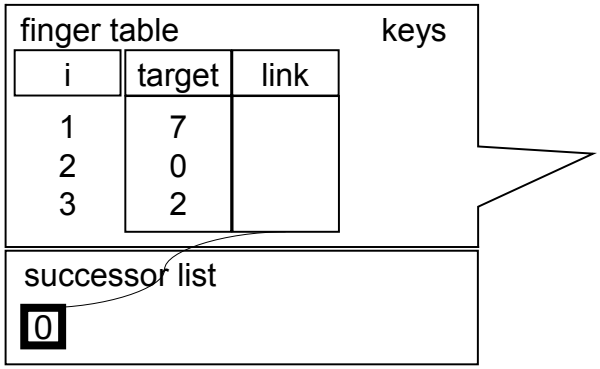


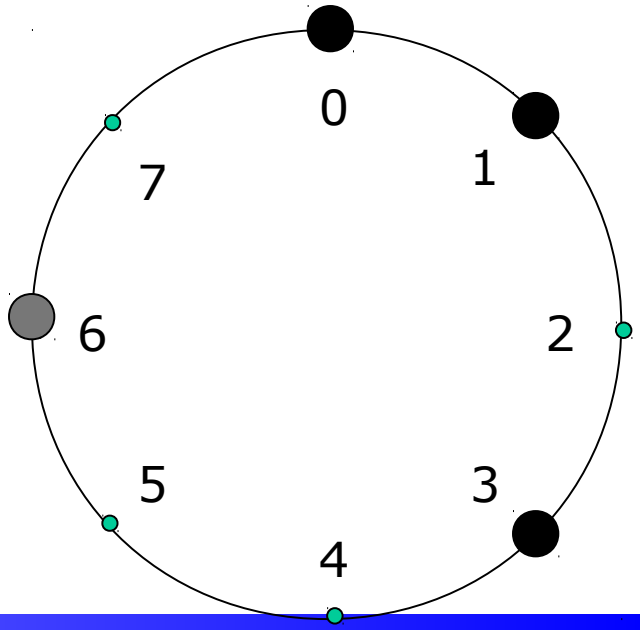

```
// Periodically at n
procedure n.stabilize() {
   succ := find first alive node in successor list
   v := succ.pred
   if (v ≠ nil and v ∈ (n,succ]) then
      set succ := v
   send a notify(n) to succ
   updateSuccessorList(succ.successorList)
}
```

CHORD: LISTA DEI SUCCESSORI

- lista dei successori di n = include i primi r successori di n sull'anello, in senso orario
- il nodo n richiede la lista dei successori al suo immediato successore s, rimuove l'ultimo elemento ed inserisce s in testa
- Se il successore cade, viene sostituito con l'elemento successivo della lista
- Look up: ricerca nella finger table + lista successori

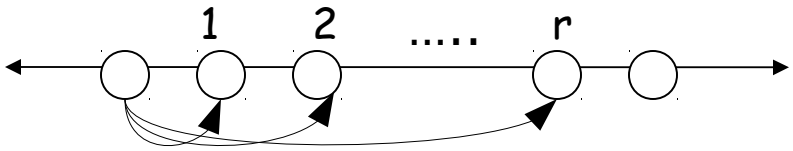


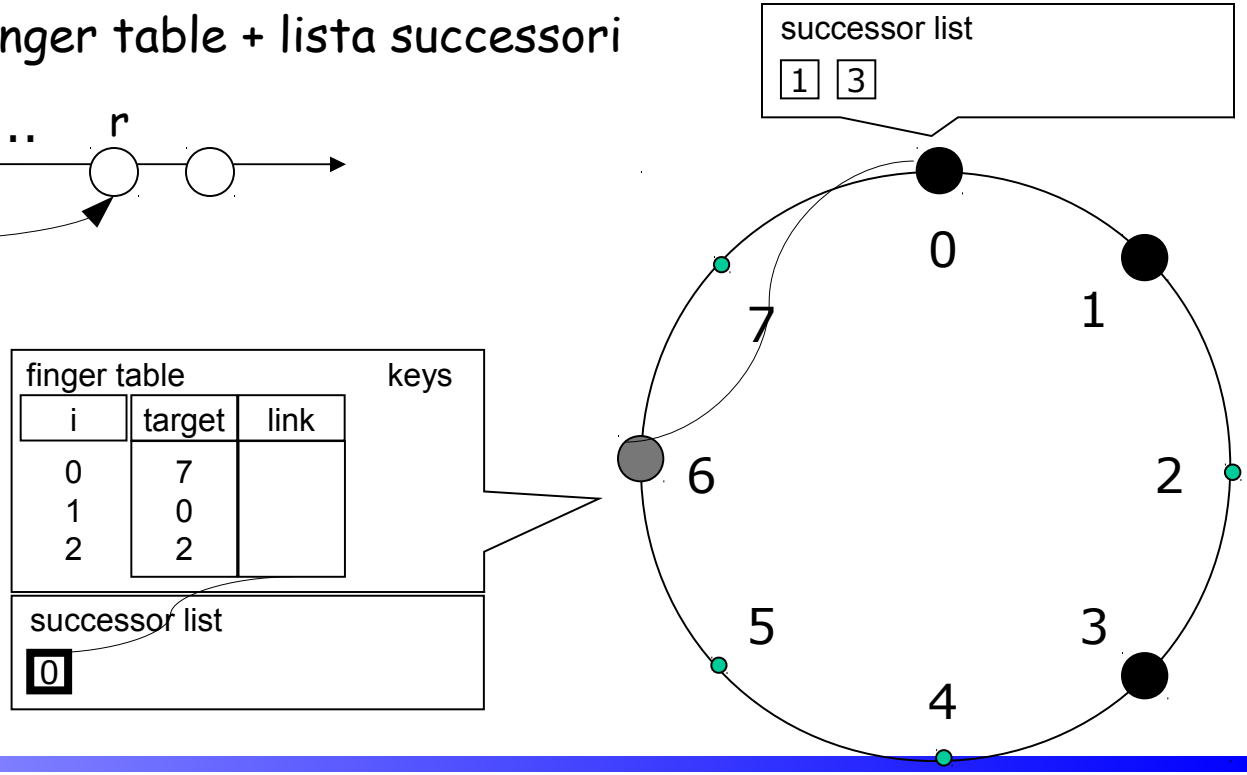




CHORD: LISTA DEI SUCCESSORI

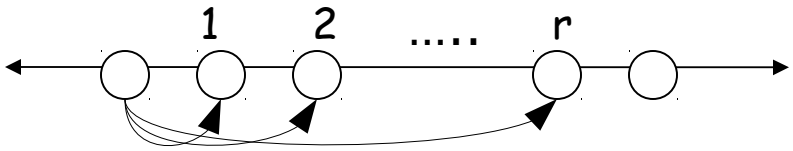
- lista dei successori di n = include i primi r successori di n sull'anello, in senso orario
- al momento dell'inserimento un nodo richiede la lista dei successori al suo immediato successore s, rimuove l'ultimo elemento ed inserisce s in testa
- Se il successore cade, viene sostituito con l'elemento successivo della lista
- Look up: ricerca nella finger table + lista successori

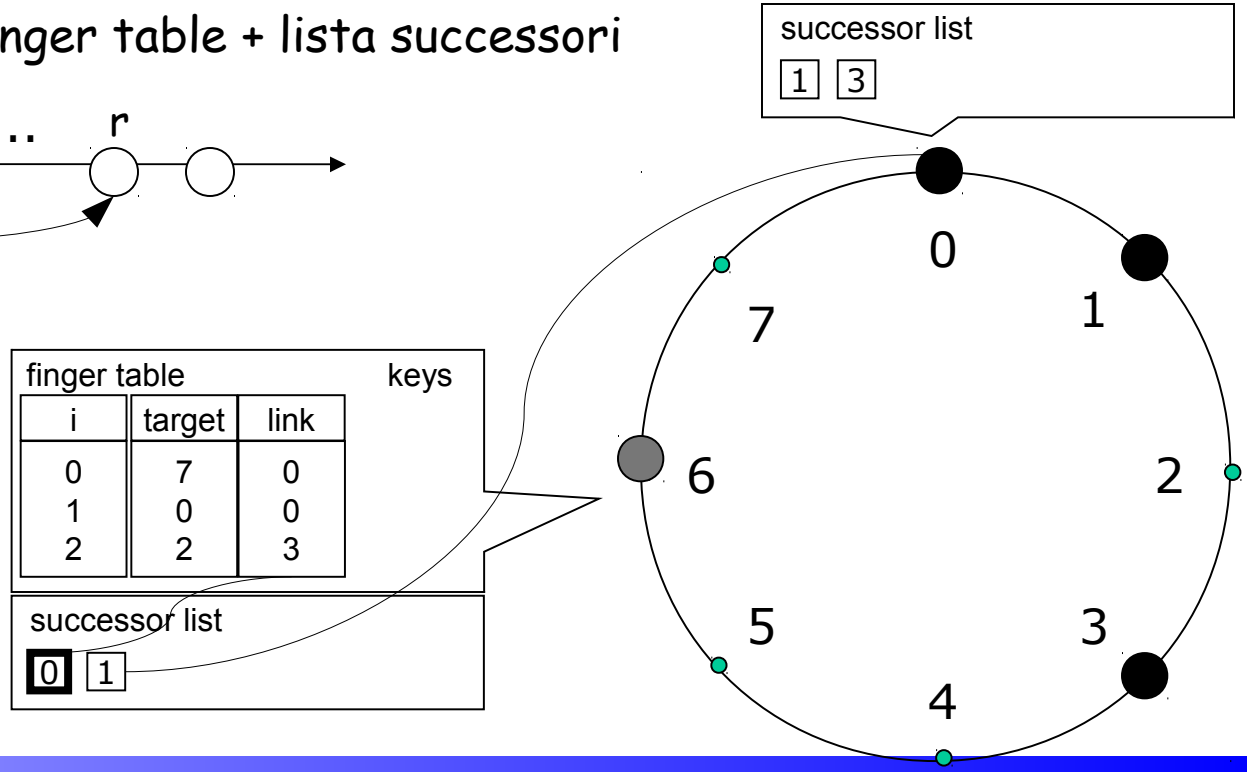




CHORD: LISTA DEI SUCCESSORI

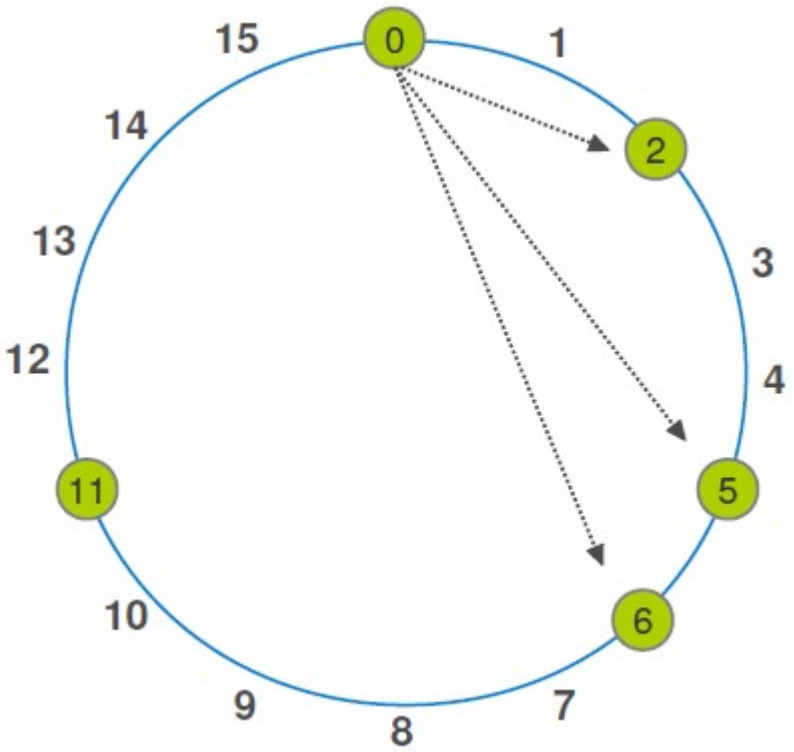
- lista dei successori di n = include i primi r successori di n sull'anello, in senso orario
- Al momento dell'inserimento, un nodo n richiede la lista dei successori al suo immediato successore s, rimuove l'ultimo elemento ed inserisce s in testa
- Se il successore cade, viene sostituito con l'elemento successivo della lista
- Look up: ricerca nella finger table + lista successori





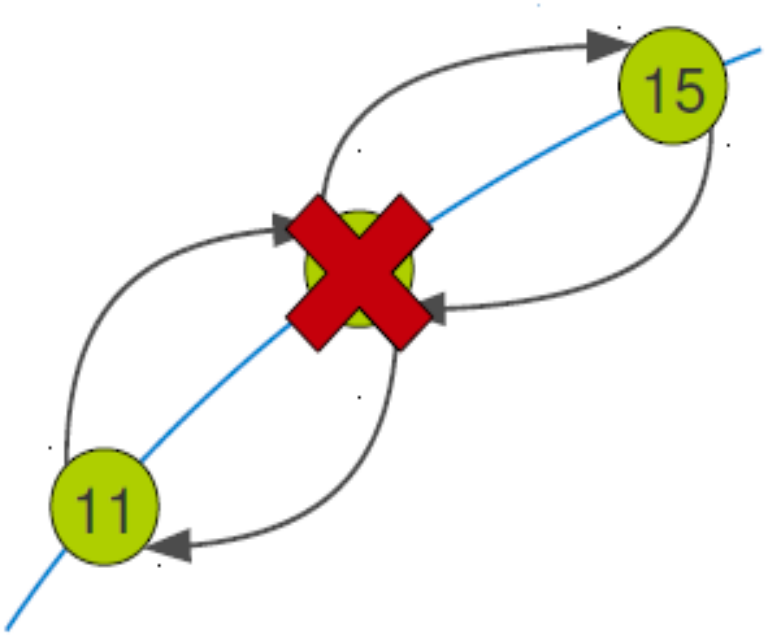
CHORD: LISTA DEI SUCCESSORI

- stabilizzazione periodica
- se il successore è fallito
  - rimpiazzare con il successore del nodo fallito, reperito dalla lista dei successori
- se il predecessore è fallito
  - settare il predecessore a nil



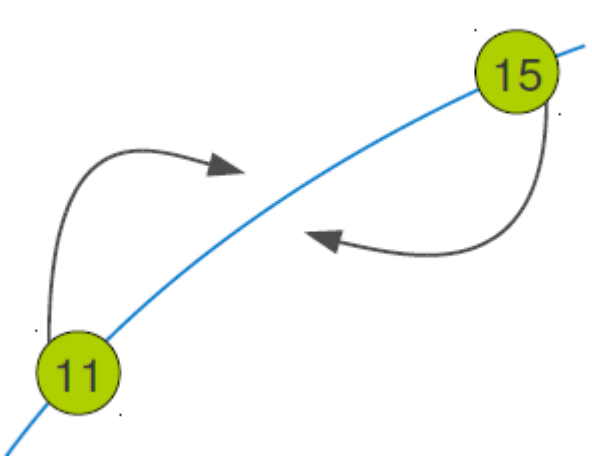
CHORD: GESTIONE DEI FALLIMENTI

- n lascia la rete in modo inaspettato
- quando succ(n) individua il fault imposta pred a nil
- quando pred(n) individua il fault imposta succ al primo successore "vivo" nella lista dei successori
- il successore individuato aggiorna prec



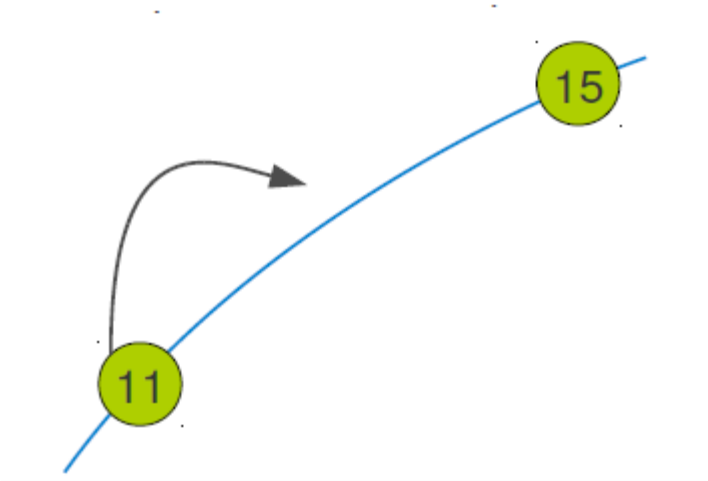
CHORD: GESTIONE DEI FALLIMENTI

- n lascia la rete in modo inaspettato
- quando succ(n) individua il fault imposta pred a nil
- quando pred(n) individua il fault imposta succ al primo successore ''vivo'' nella lista dei successori
- il successore individuato aggiorna prec



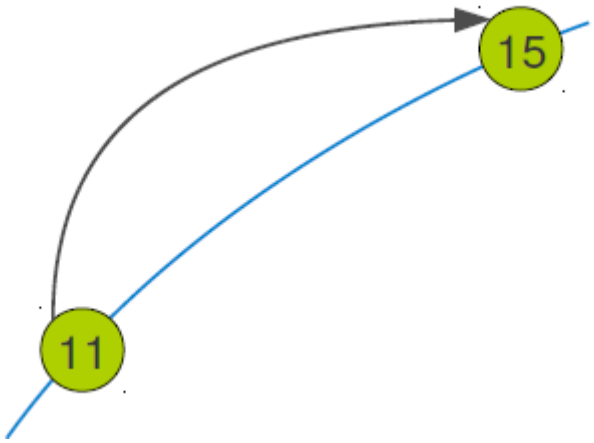
CHORD: GESTIONE DEI FALLIMENTI

- n lascia la rete in modo inaspettato
- quando succ(n) individua il fault imposta pred a nil
- quando pred(n) individua il fault imposta succ al primo successore ''vivo'' nella lista dei successori
- il successore individuato aggiorna prec



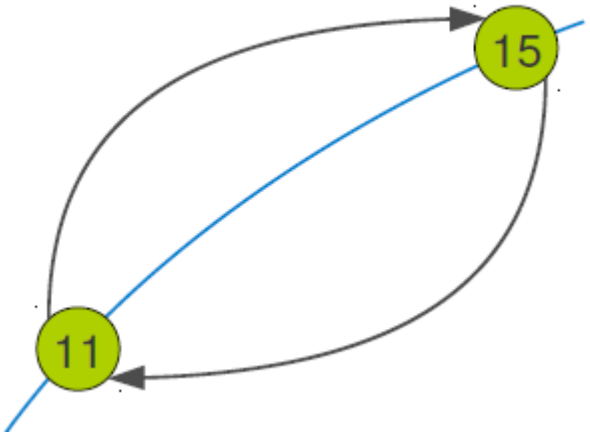
CHORD: GESTIONE DEI FALLIMENTI

- n lascia la rete in modo inaspettato
- quando succ(n) individua il fault imposta pred a nil
- quando pred individua il fault imposta succ al primo successore "vivo" nella lista dei successori
- il successore individuato aggiorna prec



CHORD: GESTIONE DEI FALLIMENTI

- n lascia la rete in modo inaspettato
- quando succ(n) individua il fault imposta pred a nil
- quando pred(n) individua il fault imposta succ al primo successore "vivo" nella lista dei successori
- il successore individuato aggiorna prec



CHORD: CADUTA DEI NODI

- La correttezza della finger table viene verificata periodicamente
  - controllo periodico della caduta dei fingers
  - sostituzione con nodi attivi
  - trade-off: traffico aggiuntivo per la gestione dei fallimenti vs. correttezza e velocità nel reperimento delle informazioni

CHORD: REPLICAZIONE DEI DATI

- Chord non garantisce l'affidabilità dei dati in presenza di fallimenti
- Ma….Chord fornisce dei meccanismi per garantirne l'affidabilità
- Ogni applicazione che utilizza il livello Chord, può utilizzare la lista dei successori di un nodo per garantire che un dato venga replicato negli r successori di un nodo
- Nel momento che un nodo fallisce, il sistema può utilizzare le repliche per individuare i dati in modo corretto

CHORD: PARTENZA VOLONTARIA DEI NODI

- Partenza volontaria di nodi
  - shutdown volontario di un nodo vs. caduta improvvisa del nodo
- Per semplicità: può essere trattata come un fault
- Ottimizzazioni: il nodo n che intende abbandonare l'anello
  - notifica la sua intenzione al successore, al predecessore ed ai fingers.
    - il predecessore può rimuovere n dalla sua lista di successori
    - il predecessore può aggiungere alla sua lista di successori il primo nodo della lista di successori di n
  - Trasferisce le chiavi di cui è responsabile al suo successore

CHORD: SCELTE DI PROGETTO

Approccio a livelli

- Chord è responsabile del routing
- La gestione dei dati è demandata alle applicazioni
  - persistenza
  - consistenza
  - fairness

Approccio soft

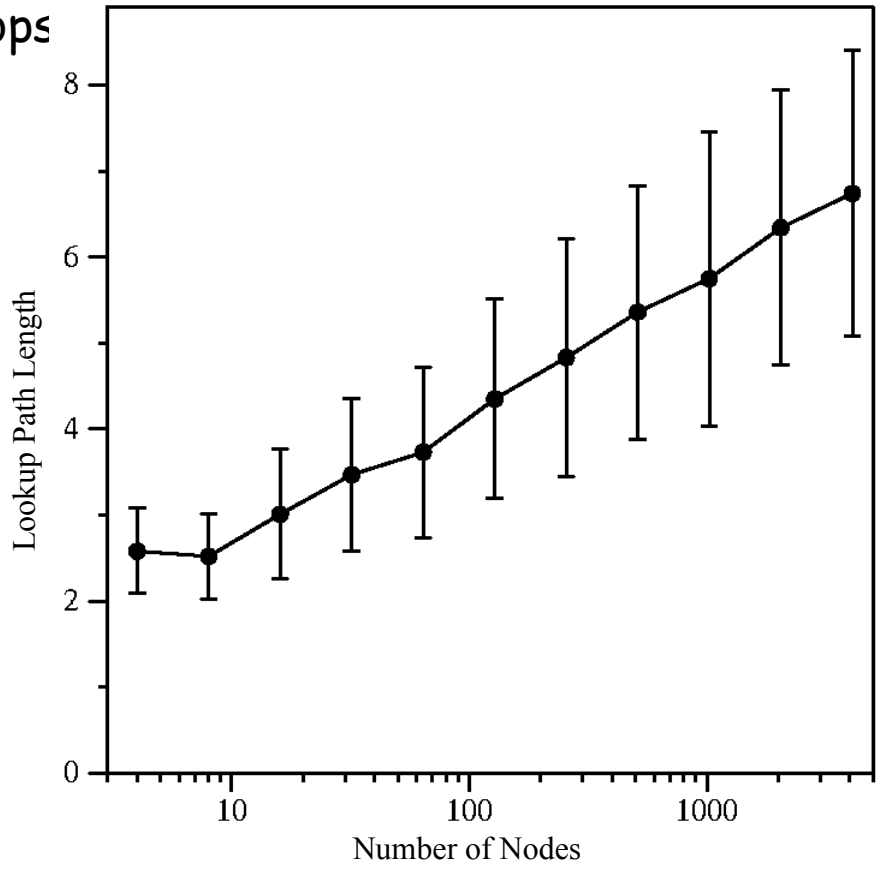
- i nodi cancellano le coppie (key, value) dopo che è trascoso un intervallo di tempo (periodo di refresh) dall'ultimo inserimento
- le applicazioni effettuano il refresh periodico delle coppie (key, value)
- in questo modo si attribuiscono le informazioni ai nuovi nodi arrivati
- se un nodo fallisce occorre aspettare il periodo di refresh per avere le informazioni nuovamente disponibili

INSERIMENTO/CADUTA DINAMICA DI NODI

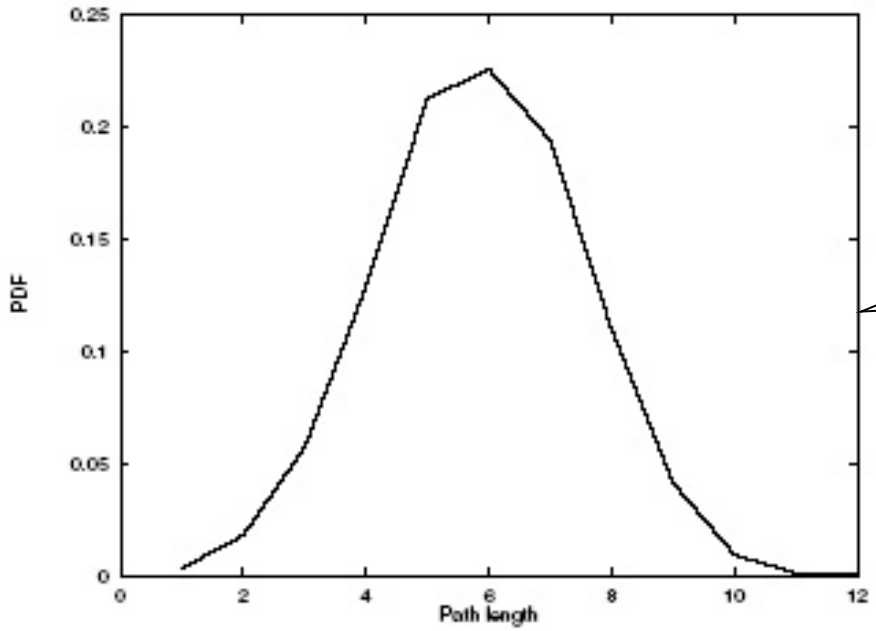
- Tutte le operazioni relative all'inserimento/caduta di un nodo sono corrette se avvengono quando l'anello si trova in uno stato stabile
- L'anello deve avere il tempo di stabilizzarsi tra due operazioni successive
- In pratica l'anello potrebbe non essersi stabilizzato prima di un nuovo inserimento/cancellazione
- Risultato generale: se i protocolli di stabilizzazione dell'anello vengono eseguiti con una frequenza opportuna, dipendente dalla frequenza degli aggiornamenti (inserimenti/fallimenti) allora l'anello rimane costantemente in uno stato stabile, in cui il routing rimane corretto e veloce (si mantiene il limite di complessità log(N))

CHORD: SIMULAZIONE

- rete = 2^k nodi, 100 x 2^k chiavi, k variabile tra 3 e 14.
- per ogni valore di k, si considera un insieme casuale di chiavi e si effettua un esperimento separato
- Per ogni chiave si valuta il numero di hops per la ricerca
- Lookup Path Length ~ $\frac{1}{2}\log_2(N)$
- Conferma dei risultati teorici
- Il grafico mostra il 1 il 99 percentile e la media



CHORD SIMULAZIONE



Lunhezza dei cammini:
PDF (Probability Density Function)
per una rete di 2^{12} nodi

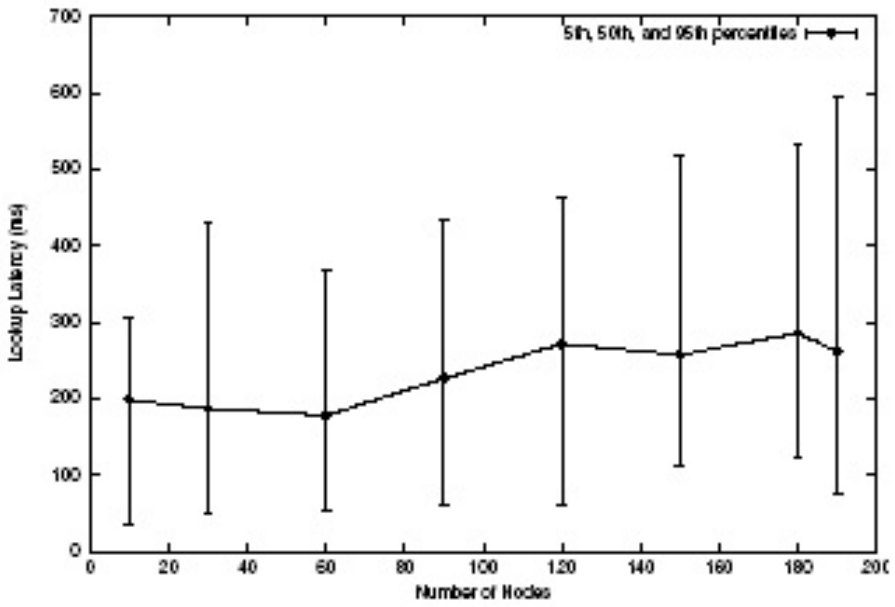
Lunghezza media dei cammini $\cong 6 = \frac{1}{2} \log_2 (2^{12})$

CHORD: IL PROTOTIPO

- Sviluppo di un prototipo Chord sviluppato su nodi Internet
- Nodi Chord dislocati in 10 siti (situati in diversi stati USA)
- Studio al variare del numero di nodi: per ogni numero di nodi sono effettuate 16 queries per chiavi scelte in modo casuale
- Latenza media varia da 180 ms. a 300 ms, dipende dal numero di nodi

CHORD: PERFORMANCE

Scalabilità
Modesto impatto del numero di nodi sulla latenza



CHORD: CONCLUSIONI

- Complessità
  - Messaggi di lookup: $O(\log N)$ hops
  - Memoria per node: $O(\log N)$ entrate nella tabella di routing
  - Messaggi per auto-organizzazione (join/leave/fail): $O(\log^2 N)$
- Vantaggi
  - Modelli teorici e prove di complessità
  - Semplice e flessibile
- Svantaggi
  - Manca una nozione di prossimità fisica
  - Casi reali: possibili situazioni limite
- Ottimizzazioni proposte
  - e.g. prossimità, links bi-direzionali, load balancing, etc.