P2P Network
Structured Networks: Distributed Hash Tables

Pedro García López

Universitat Rovira I Virgili

Pedro.garcia@urv.net

Departament d'Enginyeria
[DΣIM]
Informàtica i Matemàtiques
Universitat Rovira i Virgili

Index

- Description of CHORD’s Location and routing mechanisms
- Symphony: Distributed Hashing in a Small World

Description of CHORD's

Location and routing mechanisms

Vincent Matossian
October 12th 2001
ECE 579

Overview

Chord:

- **Maps keys onto nodes** in a 1D circular space
- Uses consistent hashing –D.Karger, E.Lehman
- Aimed at large-scale peer-to-peer applications

Talk

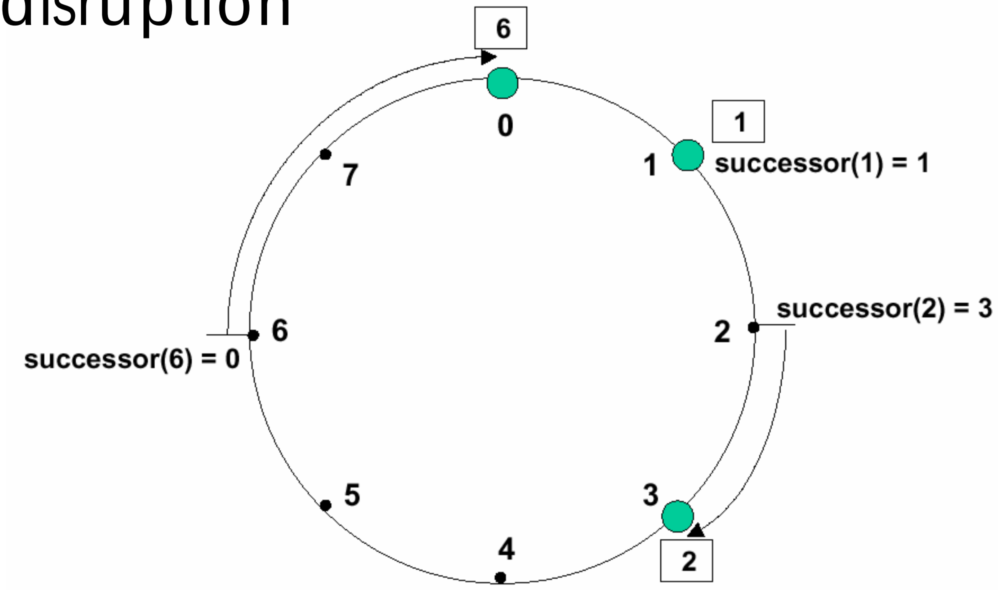
- Consistent hashing
- Algorithm for key location
- Algorithm for node joining
- Algorithm for stabilization
- Failures and replication

Consistent hashing

- Distributed caches to relieve hotspots on the web
- Node identifier hash = hash(IP address)
- Key identifier hash = hash(key)
- Designed to let nodes enter and leave the network with minimal disruption

A key is stored at its successor: node with next higher ID

In Chord hash function is Secure Hash SHA-1



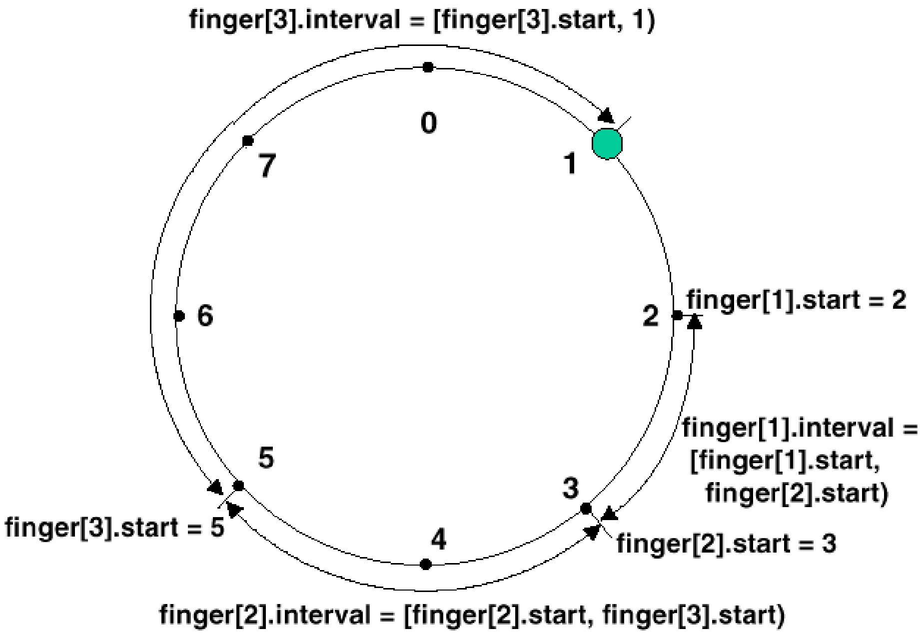
Key Location

- Finger tables allow faster location by providing additional routing information than simply successor node

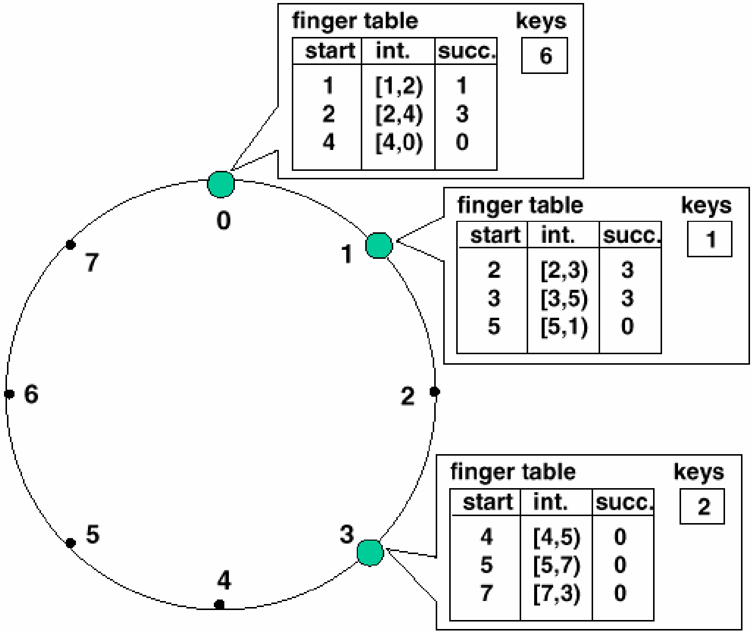
Notation	**Definition**
finger[k].start	$(n+2^{k-1}) \bmod 2^m$, 1=<k=<m
.interval	[finger[k].start,finger[k+1].start)
.node	first node>=n.finger[k].start
successor	the next node on the identifier circle; finger[1].node
predecessor	the previous node on the identifier circle

k is the finger table index

Lookup(id)



Finger table for Node 1



Node 0: finger table; keys: 6

start	int.	succ.
1	[1,2)	1
2	[2,4)	3
4	[4,0)	0

Node 1: finger table; keys: 1

start	int.	succ.
2	[2,3)	3
3	[3,5)	3
5	[5,1)	0

Node 3: finger table; keys: 2

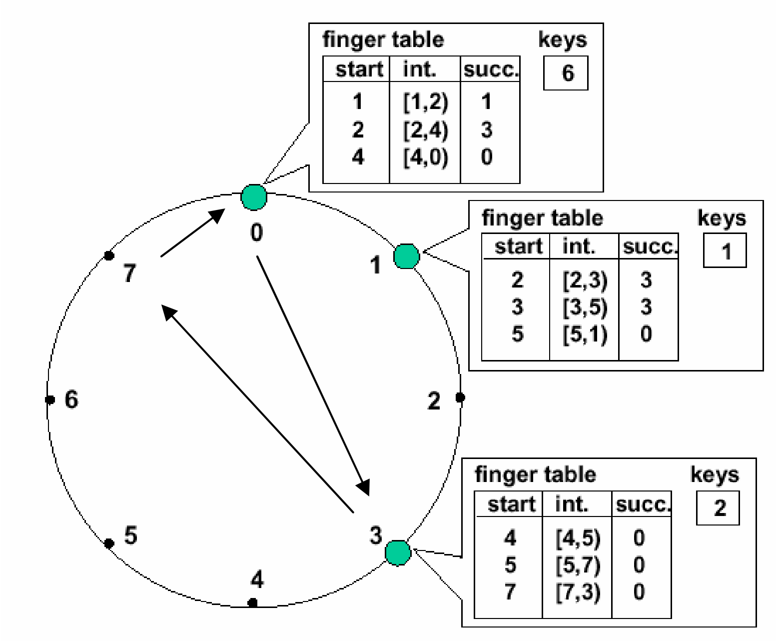
start	int.	succ.
4	[4,5)	0
5	[5,7)	0
7	[7,3)	0

Finger tables and key locations with nodes 0,1,3 and keys 1,2 and 6

Lookup PseudoCode

To find the successor of an id :

Chord returns the successor of the closest preceding finger to this id.



finger table — keys: 6

start	int.	succ.
1	[1,2)	1
2	[2,4)	3
4	[4,0)	0

finger table — keys: 1

start	int.	succ.
2	[2,3)	3
3	[3,5)	3
5	[5,1)	0

finger table — keys: 2

start	int.	succ.
4	[4,5)	0
5	[5,7)	0
7	[7,3)	0

```
// ask node $n$ to find $id$'s successor
$n$.find_successor($id$)
    $n' = find\_predecessor(id)$;
    return $n'.successor$;

// ask node $n$ to find $id$'s predecessor
$n$.find_predecessor($id$)
    $n' = n$;
    while ($id \notin (n', n'.successor]$)
        $n' = n'.closest\_preceding\_finger(id)$;
    return $n'$;

// return closest finger preceding $id$
$n$.closest_preceding_finger($id$)
    for $i = m$ downto 1
        if ($finger[i].node \in (n, id)$)
            return $finger[i].node$;
    return $n$;
```

Finding successor of identifier 1

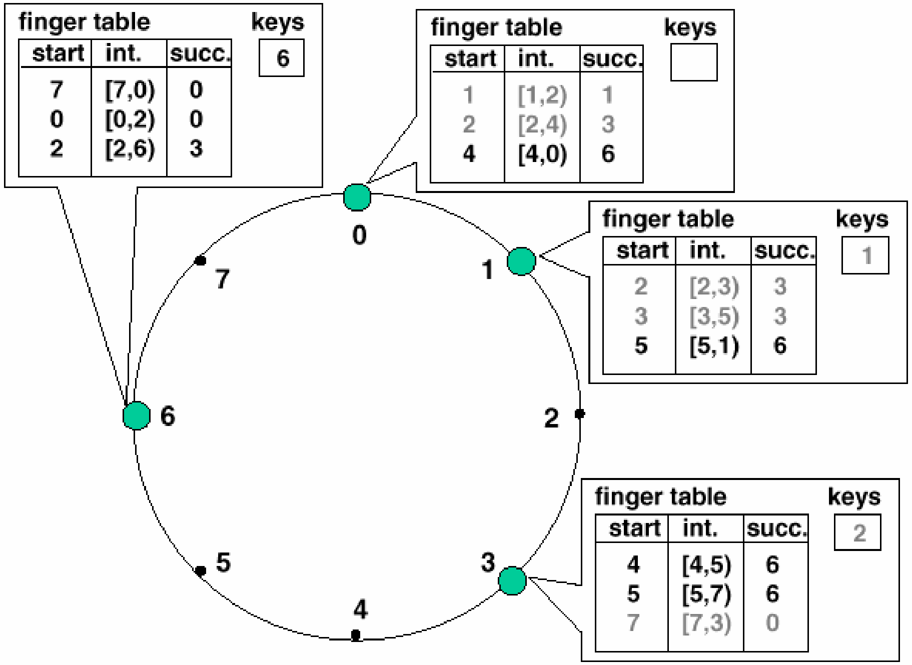
Lookup cost

- The finger pointers at repeatedly doubling distances around the circle cause each iteration of the loop in *find_predecessor* to halve the distance to the target identifier.

  In an N node Network the number of messages is of

  O(Log N)

Node Join/Leave



finger table

start	int.	succ.
7	[7,0)	0
0	[0,2)	0
2	[2,6)	3

keys
6

finger table

start	int.	succ.
1	[1,2)	1
2	[2,4)	3
4	[4,0)	6

keys

finger table

start	int.	succ.
2	[2,3)	3
3	[3,5)	3
5	[5,1)	6

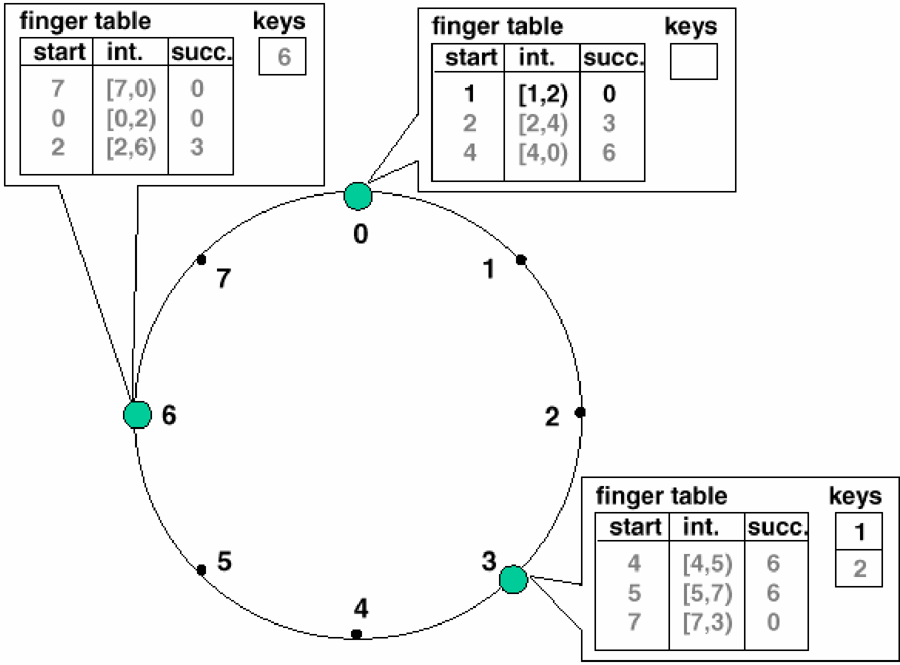
keys
1

finger table

start	int.	succ.
4	[4,5)	6
5	[5,7)	6
7	[7,3)	0

keys
2

Finger Tables and key locations after Node 6 joins



finger table

start	int.	succ.
7	[7,0)	0
0	[0,2)	0
2	[2,6)	3

keys
6

finger table

start	int.	succ.
1	[1,2)	0
2	[2,4)	3
4	[4,0)	6

keys

finger table

start	int.	succ.
4	[4,5)	6
5	[5,7)	6
7	[7,3)	0

keys
1
2

After Node 3 leaves

Changed values are in black, unchanged in gray

Join PseudoCode

Three steps:

1- **Initialize** finger and predecessor of new node n

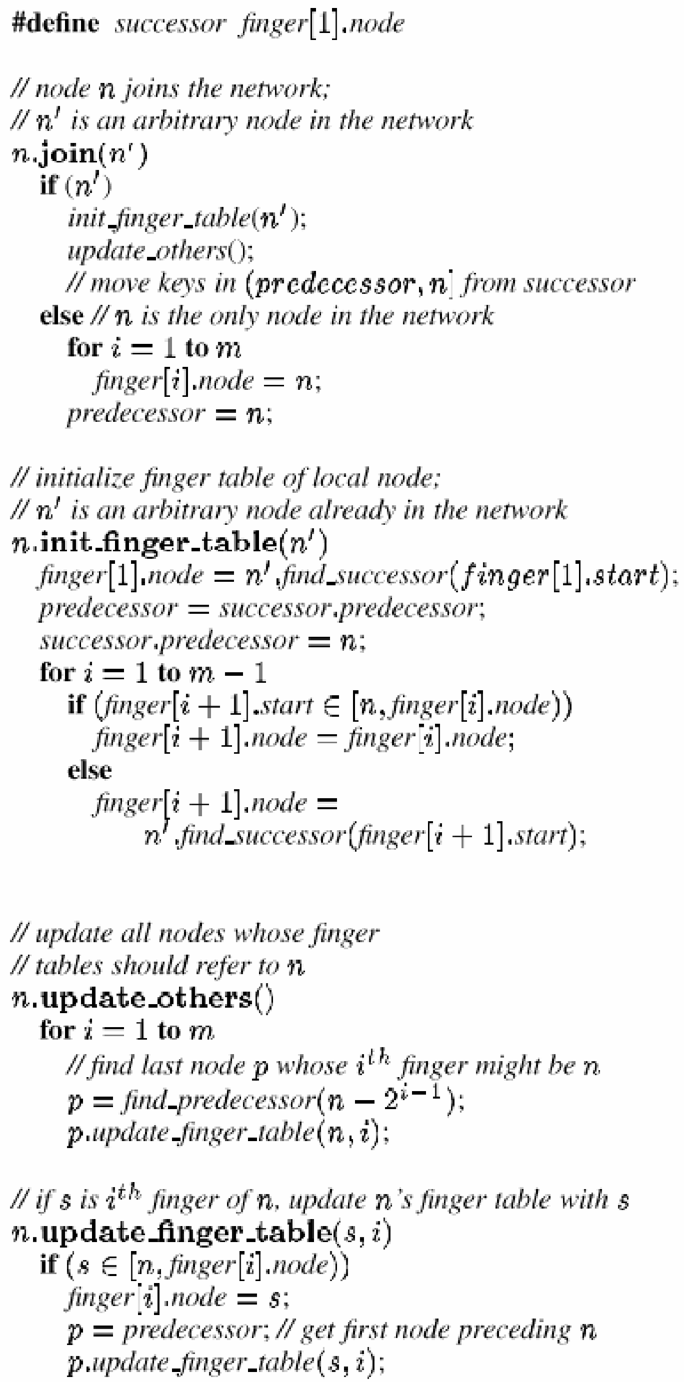
2- **Update** finger and predecessor of existing nodes to reflect the addition of n

n becomes i^{th} finger of node p if:

- p precedes n by at least 2^{i-1}
- i^{th} finger of node p succeeds n

3- **Transfer** state associated with keys that node n is now responsible for

New node n only needs to contact node that immediately forwards it to transfer responsibility for all relevant keys



```
#define successor finger[1].node

// node n joins the network;
// n' is an arbitrary node in the network
n.join(n')
  if (n')
    init_finger_table(n');
    update_others();
    // move keys in (predecessor, n] from successor
  else // n is the only node in the network
    for $i = 1$ to $m$
      finger[i].node = n;
    predecessor = n;

// initialize finger table of local node;
// n' is an arbitrary node already in the network
n.init_finger_table(n')
  finger[1].node = n'.find_successor(finger[1].start);
  predecessor = successor.predecessor;
  successor.predecessor = n;
  for $i = 1$ to $m - 1$
    if (finger[i + 1].start $\in$ [n, finger[i].node))
      finger[i + 1].node = finger[i].node;
    else
      finger[i + 1].node =
          n'.find_successor(finger[i + 1].start);


// update all nodes whose finger
// tables should refer to n
n.update_others()
  for $i = 1$ to $m$
    // find last node p whose $i^{th}$ finger might be n
    p = find_predecessor($n - 2^{i-1}$);
    p.update_finger_table(n, i);

// if s is $i^{th}$ finger of n, update n's finger table with s
n.update_finger_table(s, i)
  if (s $\in$ [n, finger[i].node))
    finger[i].node = s;
    p = predecessor; // get first node preceding n
    p.update_finger_table(s, i);
```

Join/leave cost

Number of nodes that need to be updated when a node joins is

$$O(Log\ N)$$

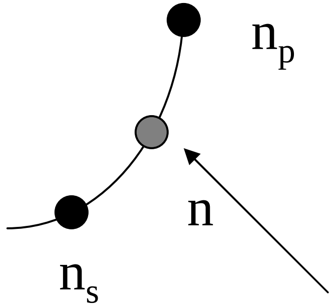
Finding and updating those nodes takes

$$O(Log^2\ N)$$

Stabilization

- If nodes join and stabilization not completed 3 cases are possible
  - finger tables are current → lookup successful
  - successors valid, fingers not → lookup successful (because find_successor succeeds) but slower
  - successors are invalid or data hasn’t migrated → lookup fails

Stabilization cont’d



Node n joins

n acquires n_s as successor

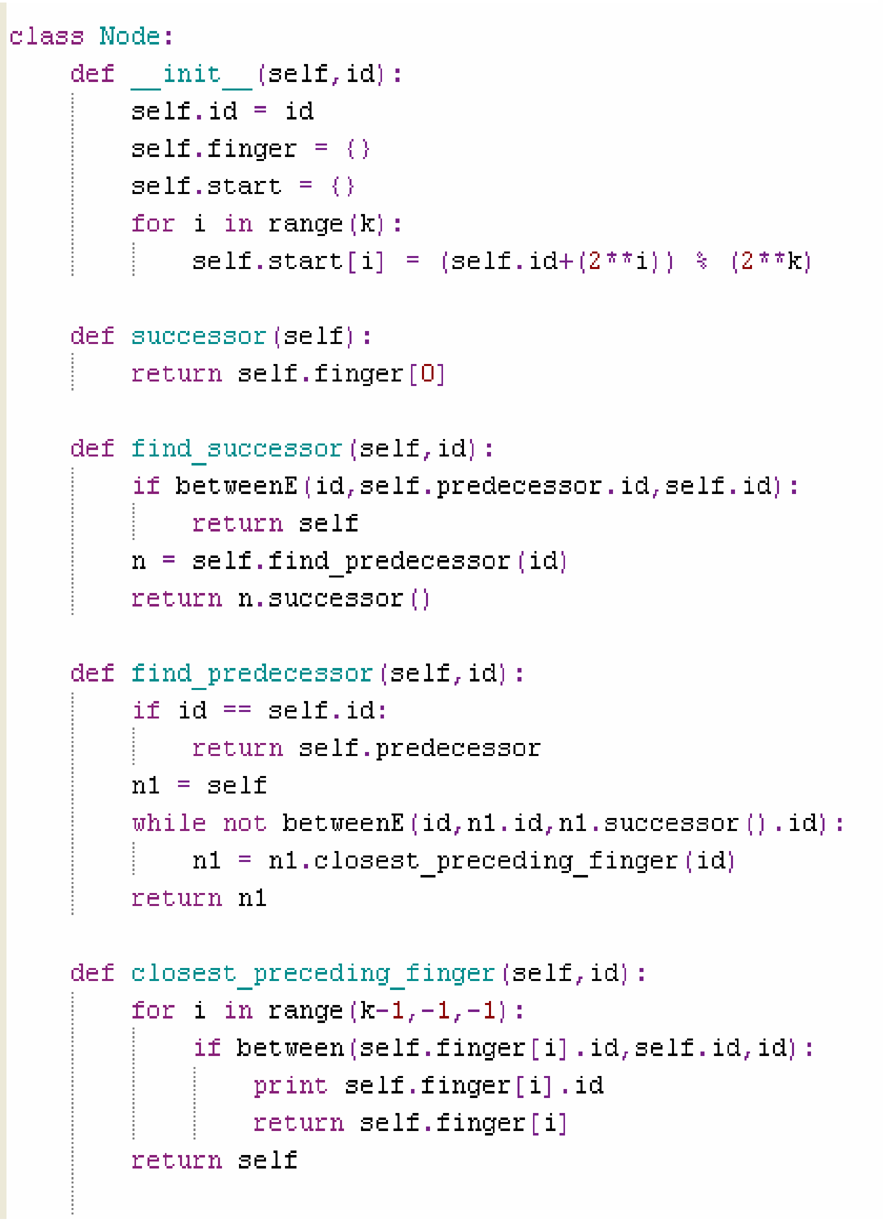
n_p runs stabilize:

- asks n_s for its predecessor (n)
- n_p acquires n as its successor
- n_p notifies n which acquires n_p as predecessor

Predecessors and successors are correct

Failures and replication

- Key step in failure recovery is correct successor pointers
- Each node maintains a successor-list of *r* nearest successors
- Knowing *r* allows Chord to inform the higher layer software when successors come and go → when it should propagate new replicas



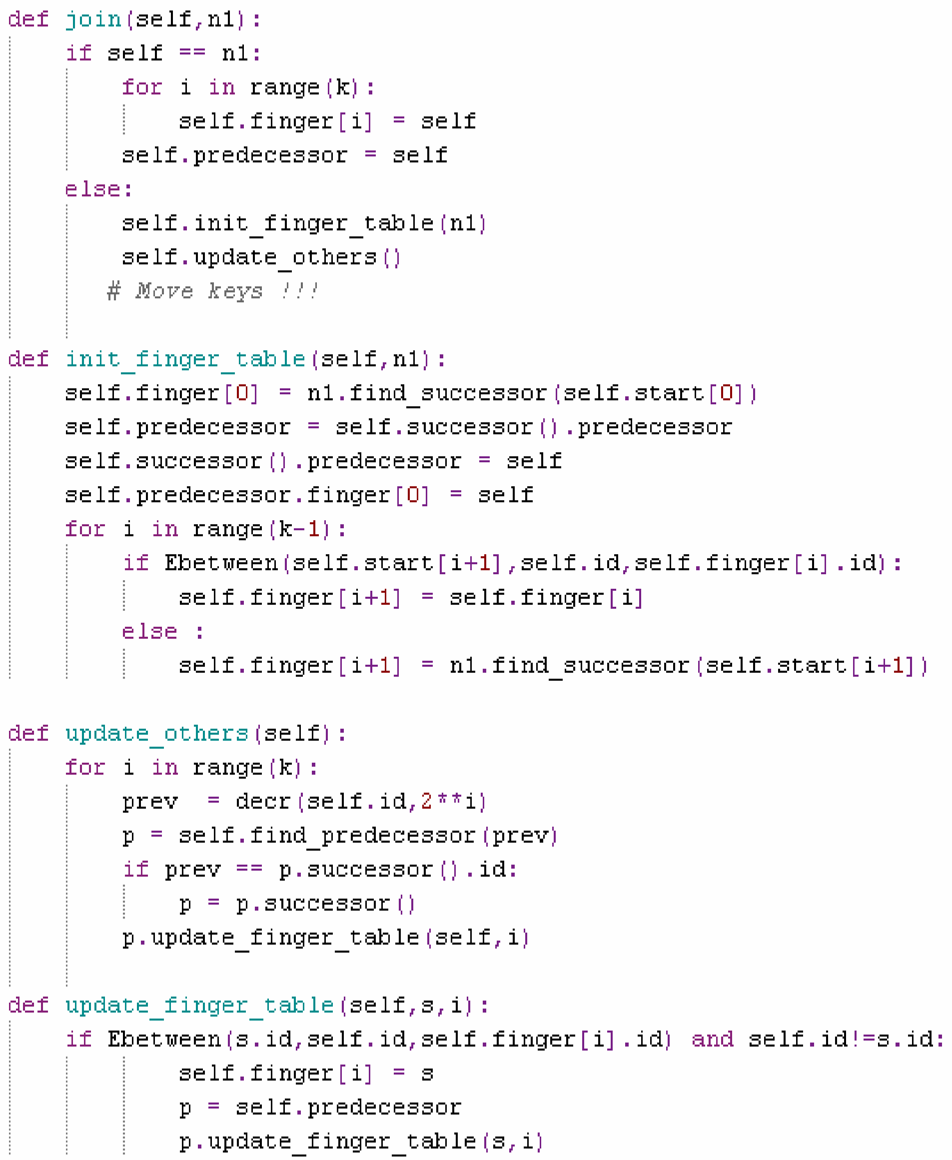
```python
class Node:
    def __init__(self,id):
        self.id = id
        self.finger = {}
        self.start = {}
        for i in range(k):
            self.start[i] = (self.id+(2**i)) % (2**k)

    def successor(self):
        return self.finger[0]

    def find_successor(self,id):
        if betweenE(id,self.predecessor.id,self.id):
            return self
        n = self.find_predecessor(id)
        return n.successor()

    def find_predecessor(self,id):
        if id == self.id:
            return self.predecessor
        n1 = self
        while not betweenE(id,n1.id,n1.successor().id):
            n1 = n1.closest_preceding_finger(id)
        return n1

    def closest_preceding_finger(self,id):
        for i in range(k-1,-1,-1):
            if between(self.finger[i].id,self.id,id):
                print self.finger[i].id
                return self.finger[i]
        return self
```



```python
    def join(self,n1):
        if self == n1:
            for i in range(k):
                self.finger[i] = self
            self.predecessor = self
        else:
            self.init_finger_table(n1)
            self.update_others()
           # Move keys !!!

    def init_finger_table(self,n1):
        self.finger[0] = n1.find_successor(self.start[0])
        self.predecessor = self.successor().predecessor
        self.successor().predecessor = self
        self.predecessor.finger[0] = self
        for i in range(k-1):
            if Ebetween(self.start[i+1],self.id,self.finger[i].id):
                self.finger[i+1] = self.finger[i]
            else :
                self.finger[i+1] = n1.find_successor(self.start[i+1])

    def update_others(self):
        for i in range(k):
            prev  = decr(self.id,2**i)
            p = self.find_predecessor(prev)
            if prev == p.successor().id:
                p = p.successor()
            p.update_finger_table(self,i)

    def update_finger_table(self,s,i):
        if Ebetween(s.id,self.id,self.finger[i].id) and self.id!=s.id:
                self.finger[i] = s
                p = self.predecessor
                p.update_finger_table(s,i)
```

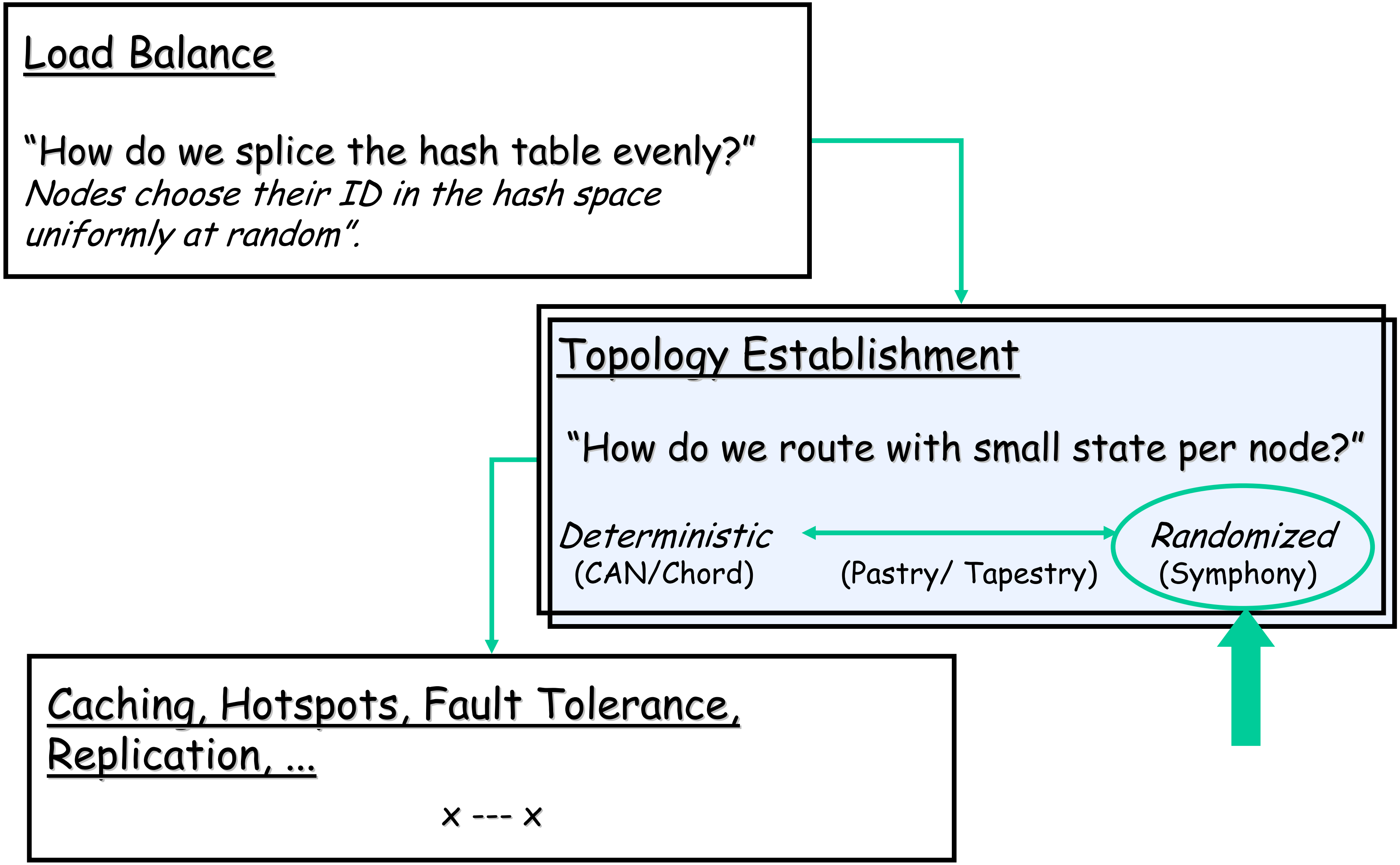
Symphony

Distributed Hashing in a Small World

Gurmeet Singh Manku
Stanford University

with **Mayank Bawa** and **Prabhakar Raghavan**

DHTs: The Big Picture



Load Balance

"How do we splice the hash table evenly?"
Nodes choose their ID in the hash space uniformly at random".

Topology Establishment

"How do we route with small state per node?"

Deterministic (CAN/Chord) ⟷ (Pastry/ Tapestry) ⟷ *Randomized* (Symphony)

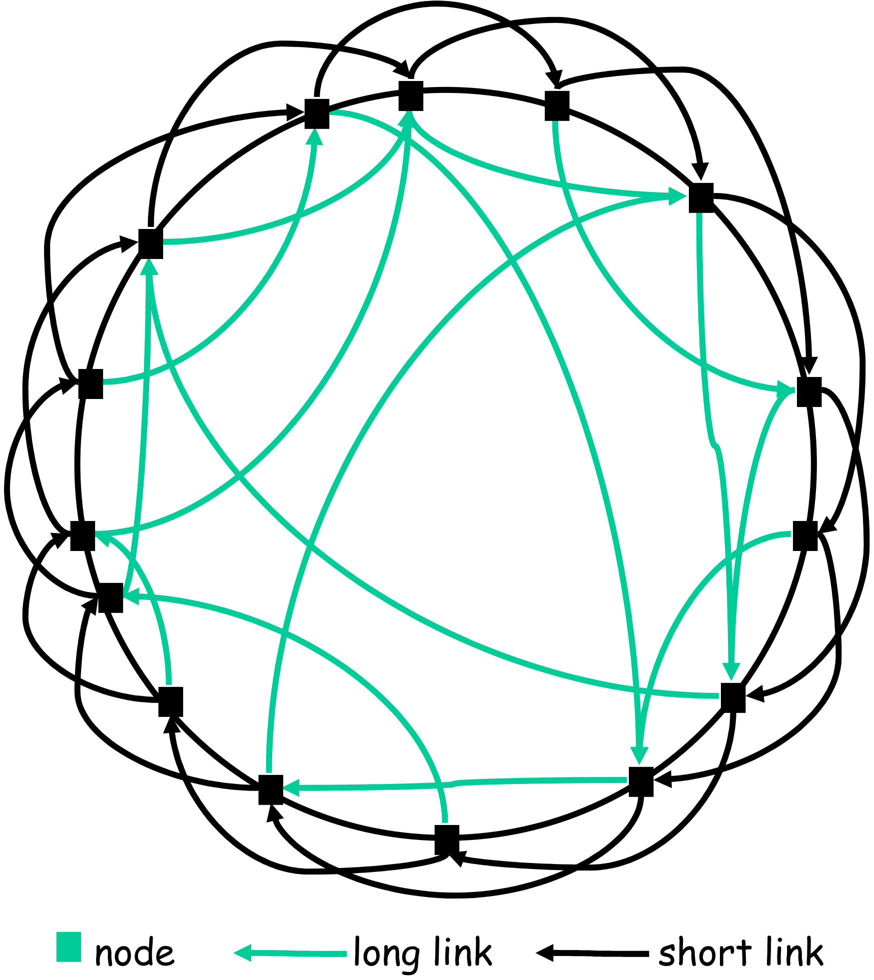
Caching, Hotspots, Fault Tolerance, Replication, ...

x --- x

Spectrum of DHT Protocols

	Protocol	#links	latency
Deterministic Topology	CAN	O(log n)	O(log n)
	Chord	O(log n)	O(log n)
Partly Randomized Topology	Viceroy	O(1)	O(log n)
	Tapestry	O(log n)	O(log n)
	Pastry	O(log n)	O(log n)
Completely Randomized Topology	Symphony	2k+2	$O((\log^2 n)/k)$

Symphony in a Nutshell



A typical Symphony network

Nodes arranged in a *unit circle* (perimeter = 1)

Arrival --> Node chooses position along circle
uniformly at random

Each node has *1 short link* (next node on circle)
and *k long links*

Adaptation of Small World Idea: [Kleinberg00]
Long links chosen from a probability distribution function: $p(x) = 1 / x \log n$ where n = #nodes.

?

Simple greedy routing:

"Forward along that link that minimizes the absolute distance to the destination."

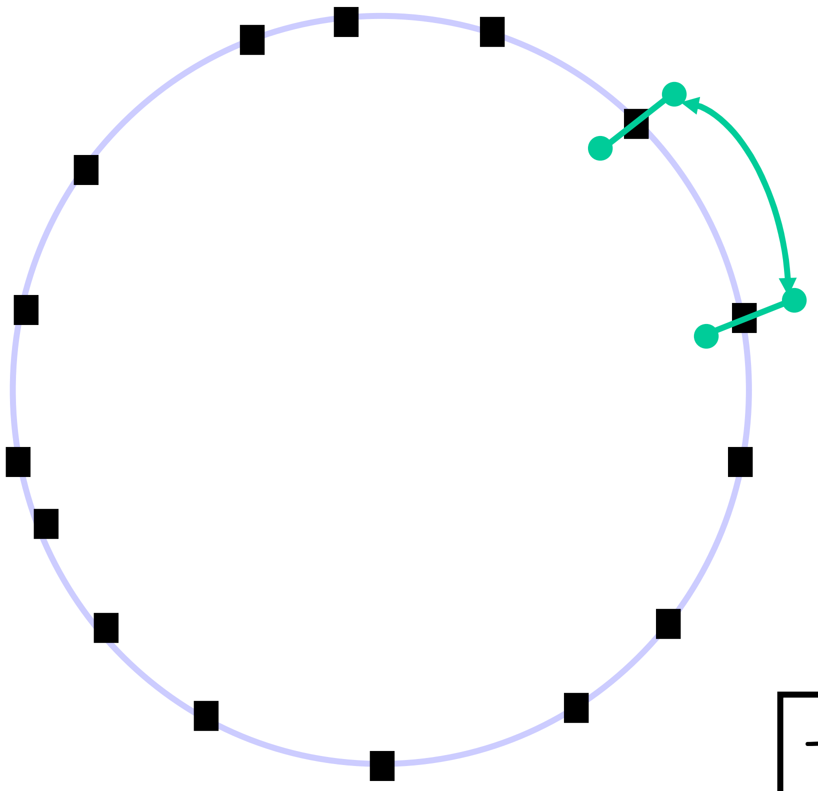
Average lookup latency = $O((\log^2 n) / k)$ hops

Fault Tolerance:

No backups for long links! Only short links are fortified for fault tolerance.

Network Size Estimation Protocol

Problem: What is the current value of n, the total number of nodes?



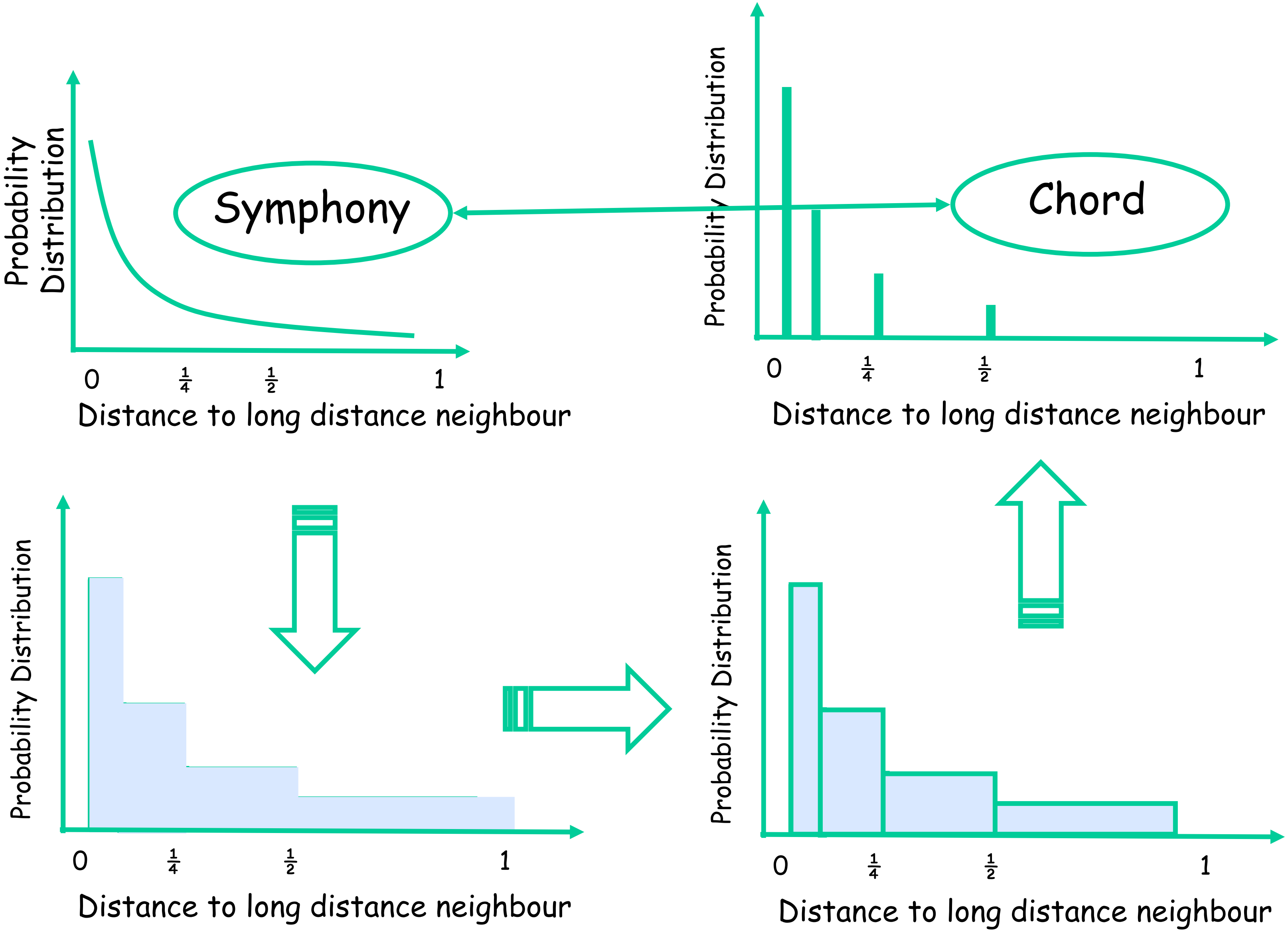
x = Length of arc

1/x = Estimate of n

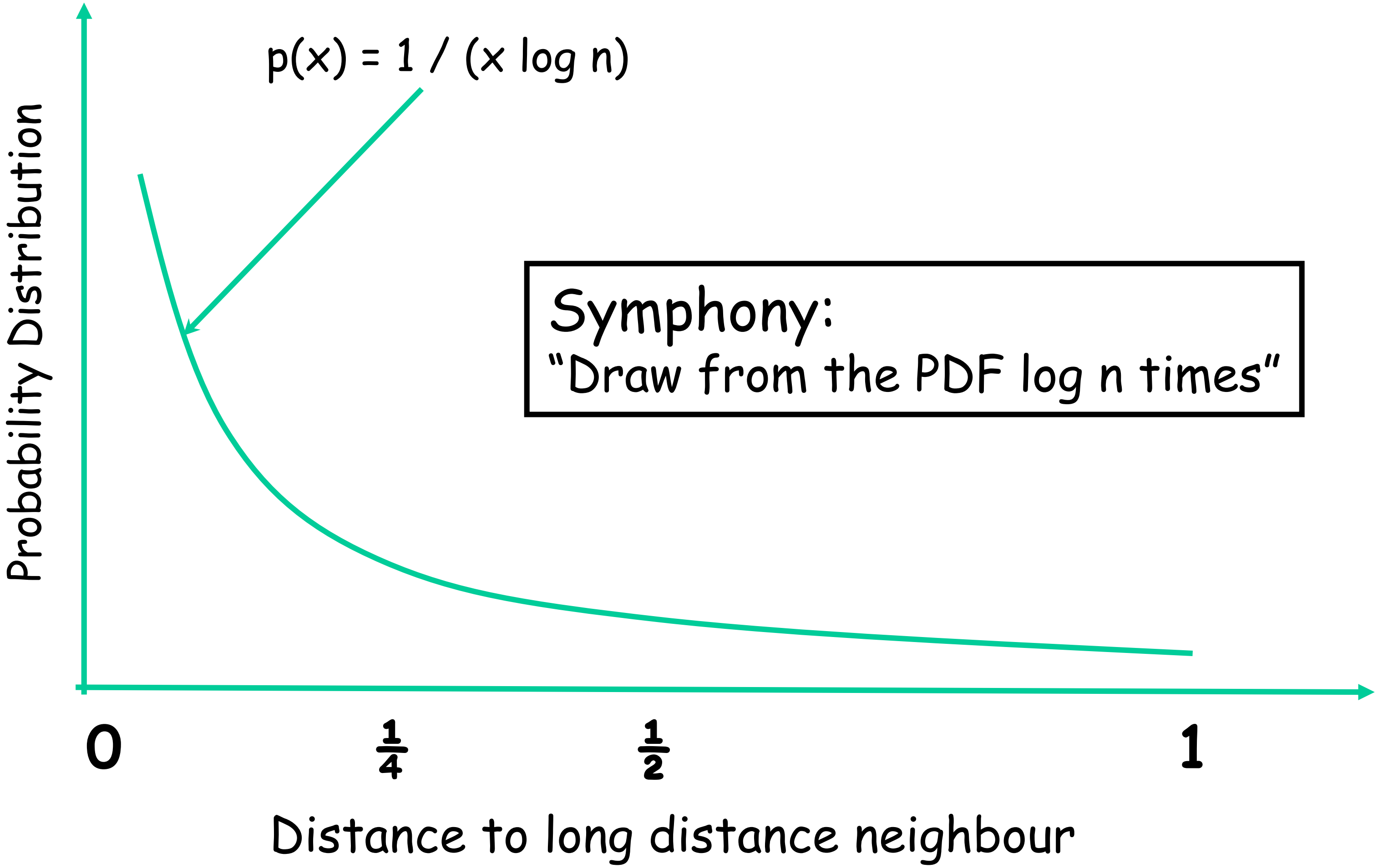
(Idea from Viceroy)

- 3 arcs are enough.
- Re-linking Protocol not worthwhile.

Intuition Behind Symphony’s PDF

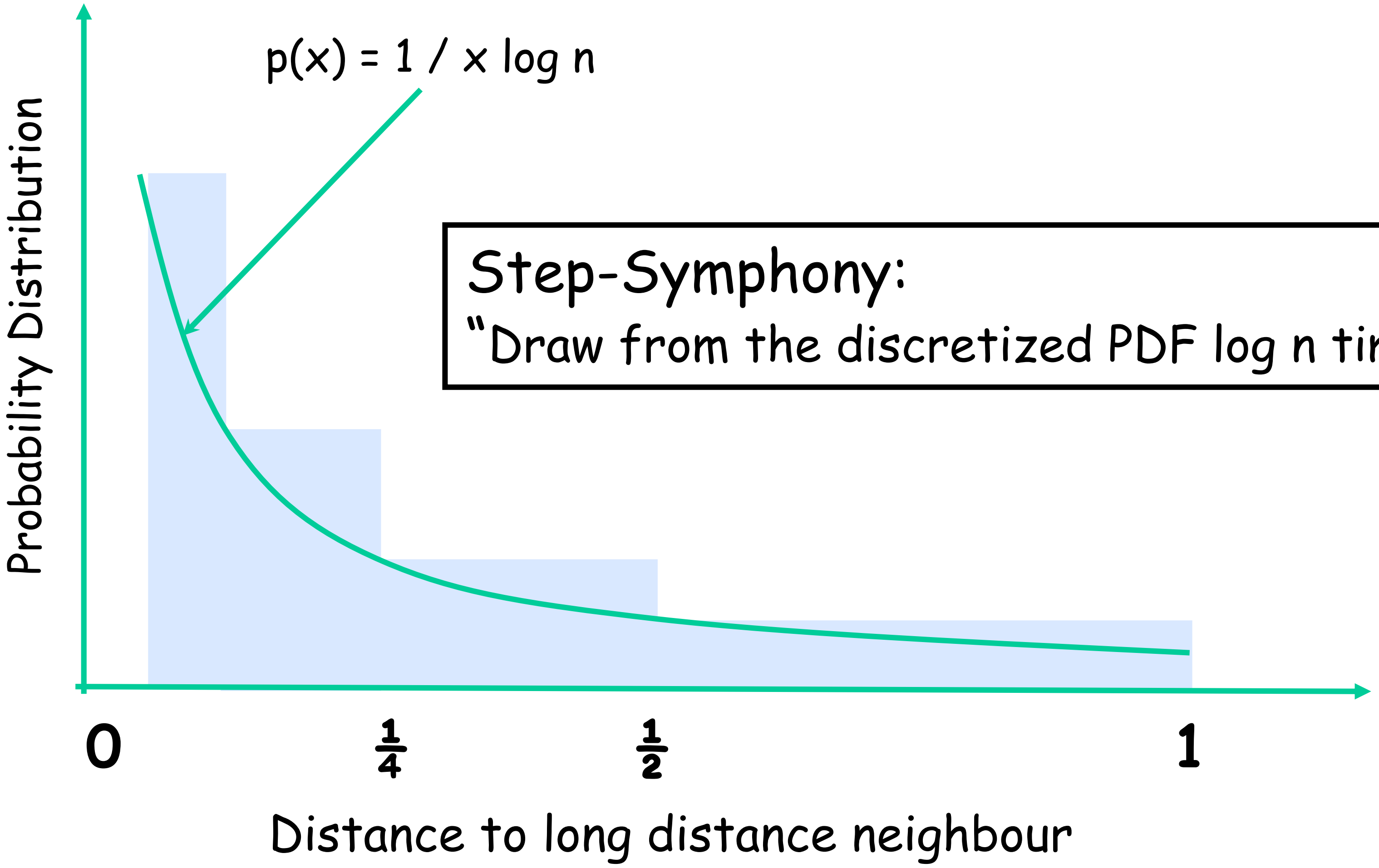


Step 0: Symphony



Symphony:
"Draw from the PDF log n times"

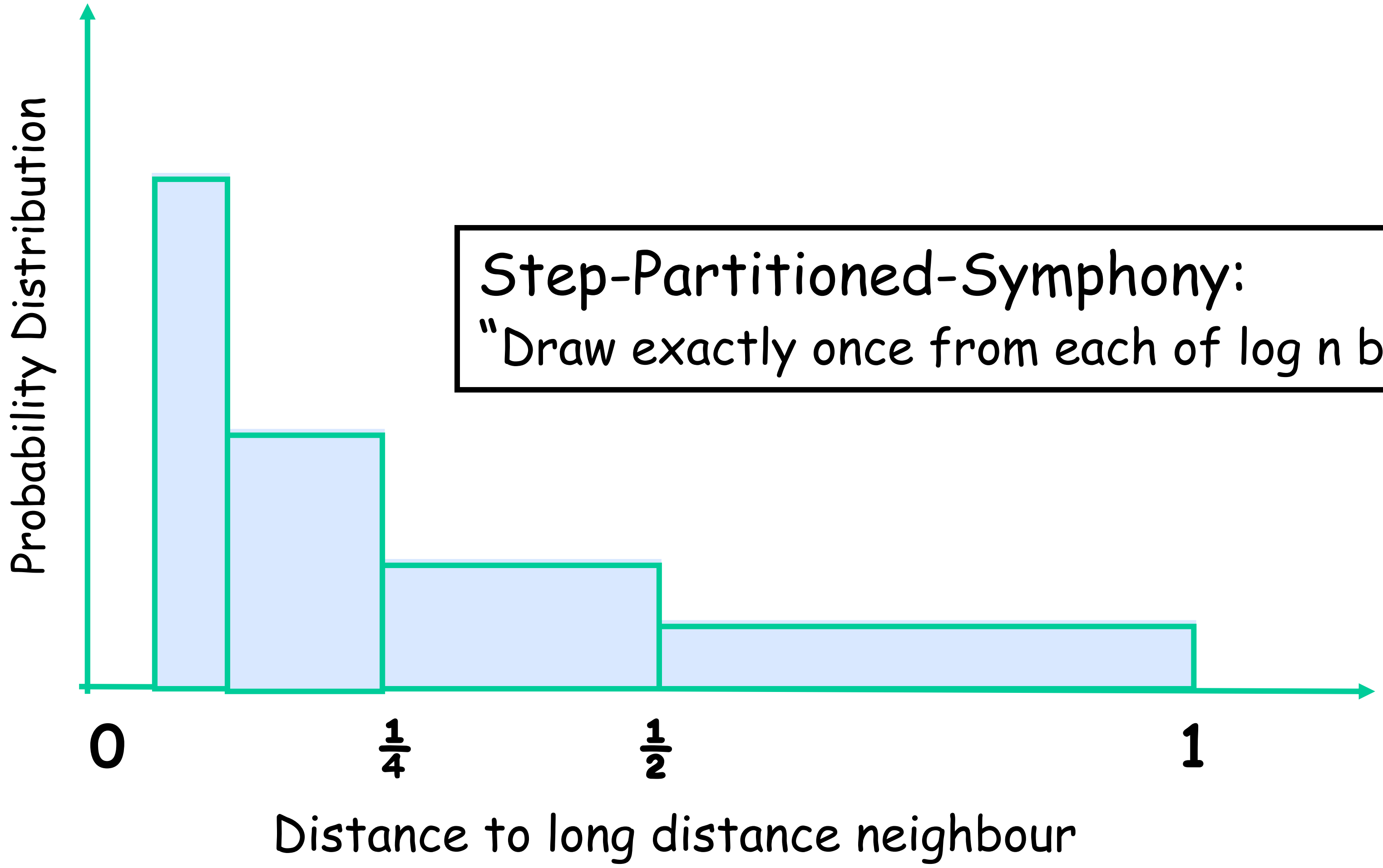
Step 1: Step-Symphony



Step-Symphony:

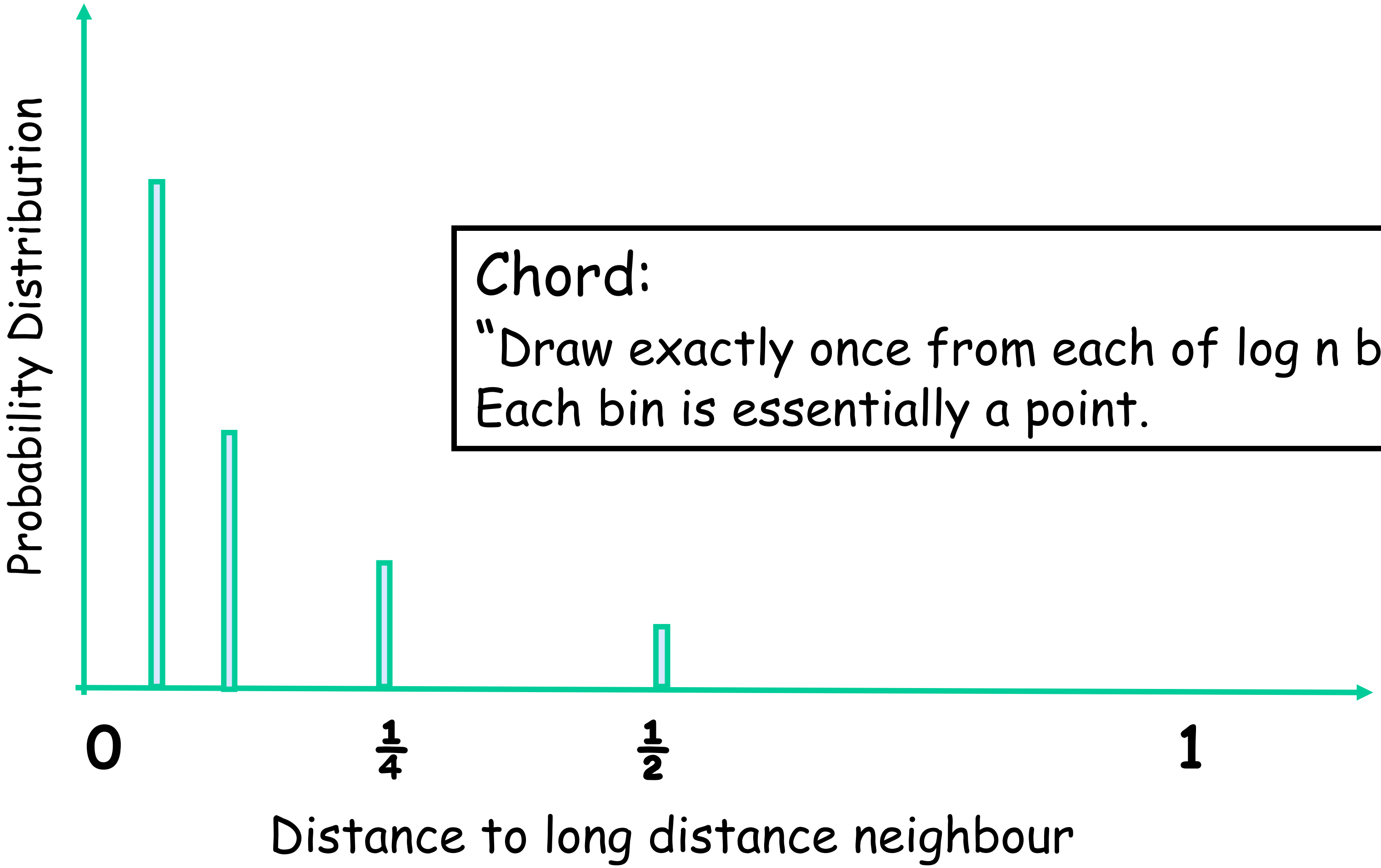
"Draw from the discretized PDF log n times"

Step 2: Divide PDF into log n Equal Bins



Step-Partitioned-Symphony:
"Draw exactly once from each of log n bins"

Step 3: Discrete PDF



Chord:

"Draw exactly once from each of log n bins"

Each bin is essentially a point.

Two Optimizations

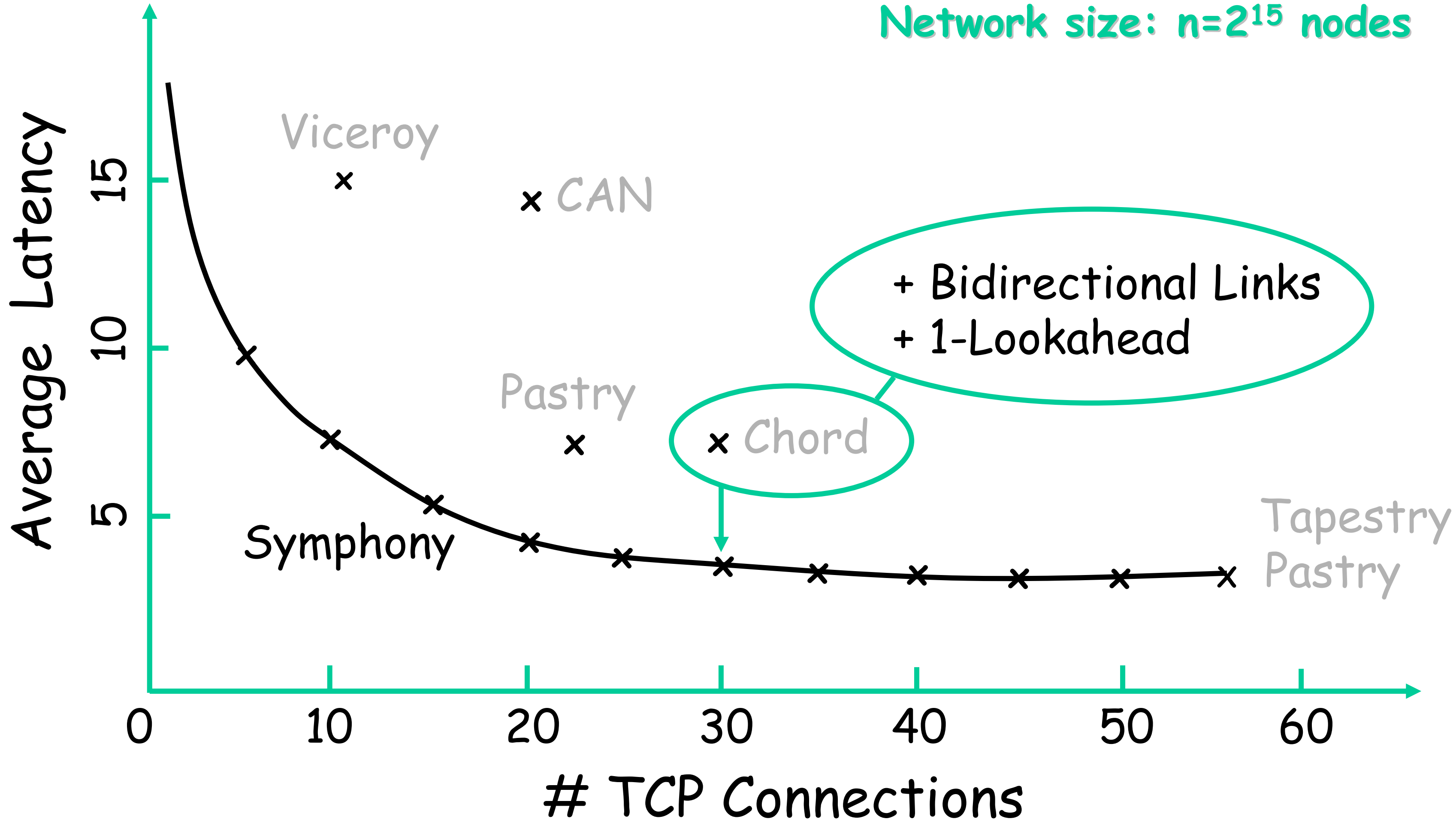
Bi-directional Routing

- Exploit both outgoing and incoming links!
- Route to the neighbor that minimizes absolute distance to destination
- Reduces avg latency by 25-30%

1-Lookahead

- List of neighbor's neighbors
- Reduces avg latency by 40%

Latency vs State Maintenance



Many more graphs in the paper.

Why Symphony?

1. Low state maintenance

Low degree --> Fewer pings/keep-alives, less control traffic
Low degree --> Distributed locking and coordination overhead over smaller sets of nodes
Low degree --> Smaller bootstrapping time when a node joins
Smaller recovery time when a node leaves

2. Fault tolerance

Only short links are bolstered. No backups for long links !

3. Smooth out-degree vs latency tradeoff

Only protocol that offers this tuning knob even at run time!
Out-degree is not fixed at runtime, or as a function of network size.

4. Flexibility and support for heterogeneity

Different nodes can have different #links !