Peer-to-Peer Computing and Structured Overlay Network

Dhruvi Mehta – 19125010

Veer Shah – 19125052

Introduction

- **Peer-to-peer (P2P) network:** application-level organization of the network overlay for flexibility sharing resources (e.g., files and multimedia documents) stored across network-wide computers.
- All nodes (called **peers**) are equal i.e. client as well as server; communication directly between peers (**no client-server**)
- Allows to find location of arbitrary objects; **no DNS servers** required
- Sharing of **Large combined storage**, CPU power, other resources, without scalability costs
- **Dynamic insertion and deletion of nodes** called churn, as well as of resources, at low cost
- **Overlay networks:** Overlay networks refer to networks that are constructed on top of another network (e.g. IP).
- **P2P overlay network:** Any overlay network that is constructed by the Internet peers in the application layer on top of the IP network.

Desirable Characteristics of P2P

Features	Performance
Self-organizing	Large combined storage, CPU power, and resources
Distributed control	Fast search for machines and data objects
Role symmetry for nodes	Scalable
Anonymity	Efficient management of churn
Naming mechanism	Selection of geographically close servers
Security, authentication, trust	Redundancy in storage and paths

Table I: Desirable characteristics and performance features of P2P systems.

Application Layer Overlays

- A core mechanism in **P2P networks is searching for data**, and this mechanism depends on how (i) the data, and (ii) the network, are organized.
- **Search algorithms** for P2P networks tend to be **data-centric**, as opposed to the **host-centric** algorithms for traditional networks.
- P2P search uses the P2P overlay, which is a **logical graph among the peers** that is used for the object search and object storage and management algorithms.
- Note that above the P2P overlay is the application layer overlay, where communication between peers is point-to-point (representing a logical all-to-all connectivity) once a connection is established.

Classification of P2P Overlay Network

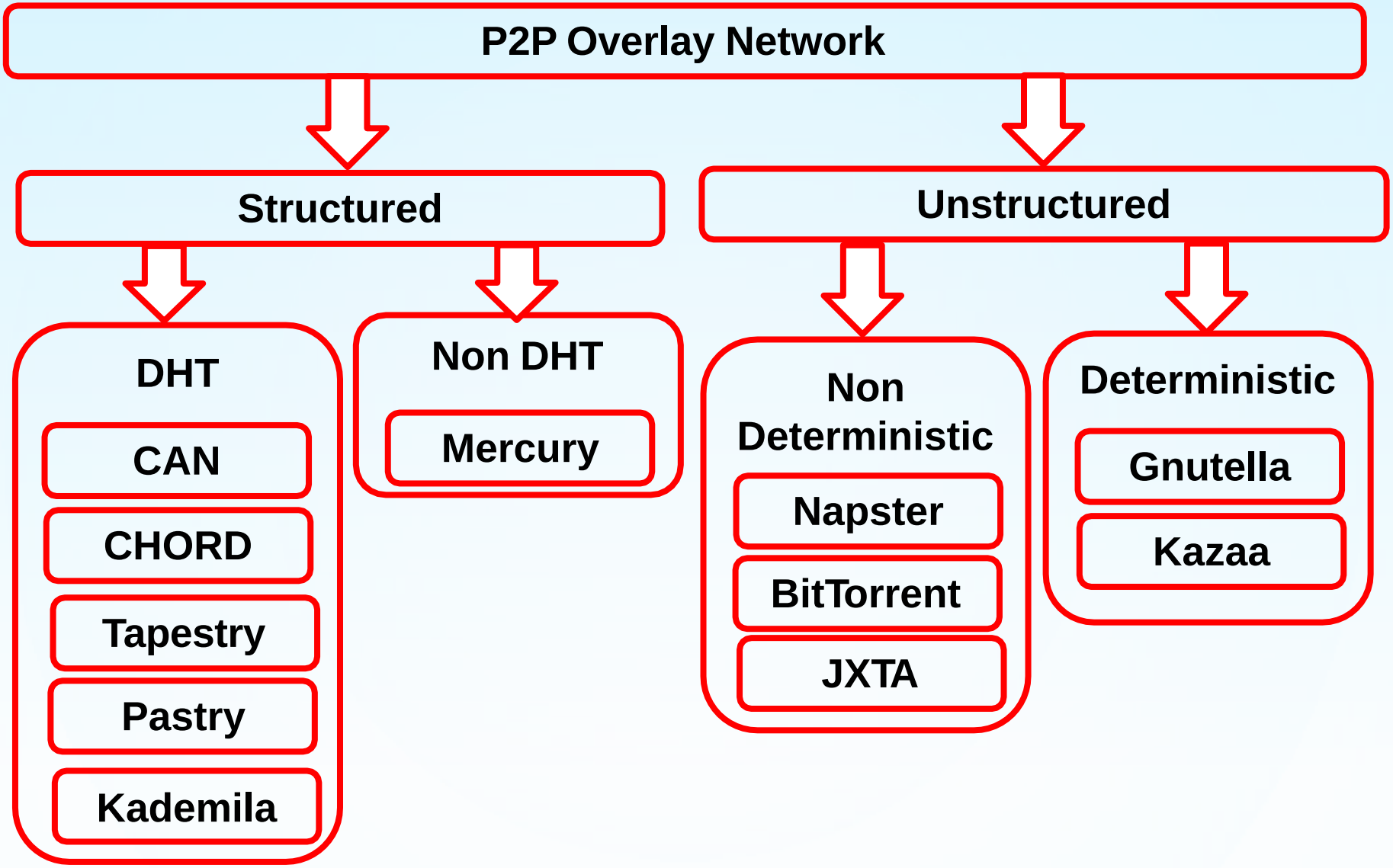


Figure: Classification of P2P Overlay Network

Classification of P2P Overlay Network

- The P2P overlay can be (i) **structured** (e.g., hypercubes, meshes, butterfly networks, de Bruijn graphs) or (ii) **unstructured**, i.e., no particular graph structure is used.

(i)Structured overlays use some rigid organizational principles based on the properties of the P2P overlay graph structure, for the object storage algorithms and the object search algorithms.

(ii)Unstructured overlays use very loose guidelines for object storage. As there is no definite structure to the overlay graph, the search mechanisms are more "ad-hoc," and typically use some forms of flooding or random walk strategies.

- Thus, object storage and search strategies are intricately linked to the overlay structure as well as to the data organization mechanisms.

Structured vs. Unstructured Overlays

Structured Overlays:

- Structure ⇒ placement of files is highly deterministic, file insertions and deletions have some overhead
- Fast lookup
- Hash mapping based on a single characteristic (e.g., file name)
- Range queries, keyword queries, attribute queries difficult to support
- **Examples:** Chord, Content Addressable Network(CAN), Pastry.

Unstructured Overlays:

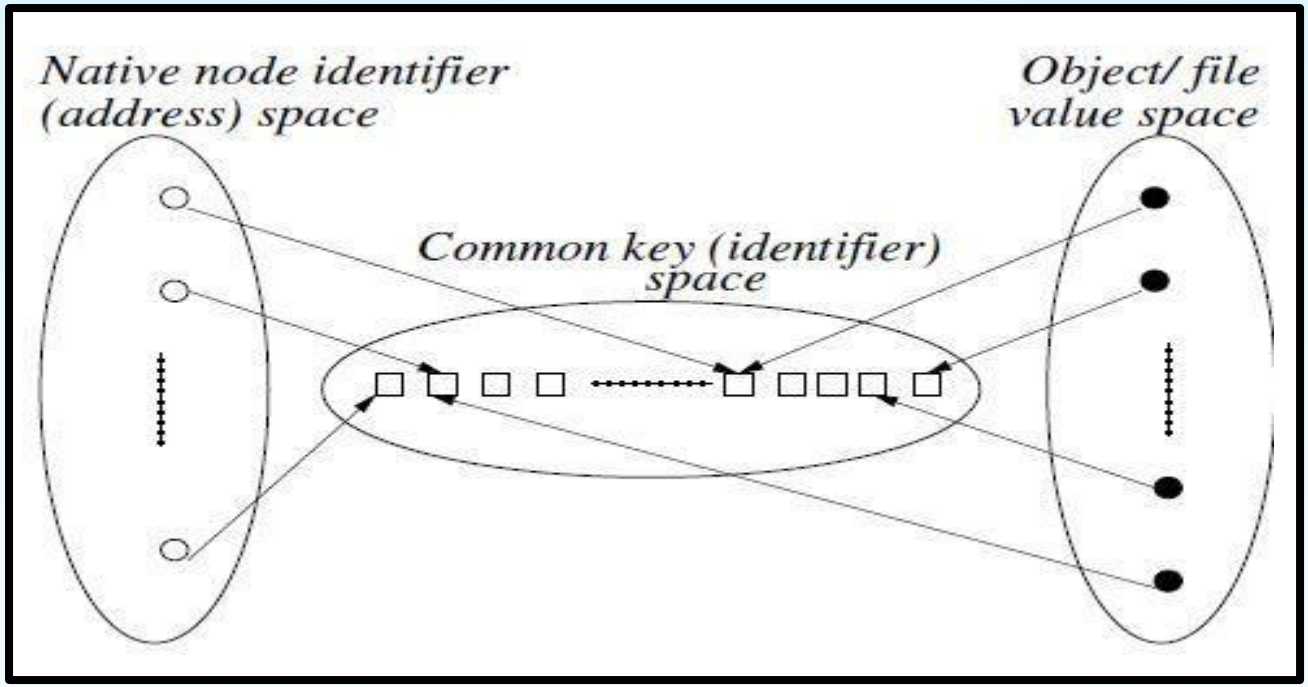
- No structure for overlay ⇒ no structure for data/file placement
- Node join/departures are easy; local overlay simply adjusted
- Only local indexing used
- File search entails high message overhead and high delays
- Complex, keyword, range, attribute queries supported
- **Examples:** FreeNet, Gnutella, KaZaA, BitTorrent

Data Indexing

- Data identified by indexing, which allows physical data independence from apps.
  - **Centralized indexing:** e.g., versions of Napster, DNS
  - **Distributed indexing:** Indexes to data scattered across peers. Access data through mechanisms such as Distributed Hash Tables (DHT). These differ in hash mapping, search algorithms, diameter for lookup, fault tolerance, churn resilience.
  - **Local indexing:** Each peer indexes only the local objects. Remote objects need to be searched for. Typical DHT uses flat key space. Used commonly in unstructured overlays (E.g., Gnutella) along with flooding search or random walk search.
- **Another classification:**
  - **Semantic indexing:** human readable, e.g., filename, keyword, database key. Supports keyword searches, range searches, approximate searches.
  - **Semantic-free indexing**: Not human readable. Corresponds to index obtained by use of hash function.

Structured Overlays: Distributed Hash Tables

Simple Distributed Hash Table scheme



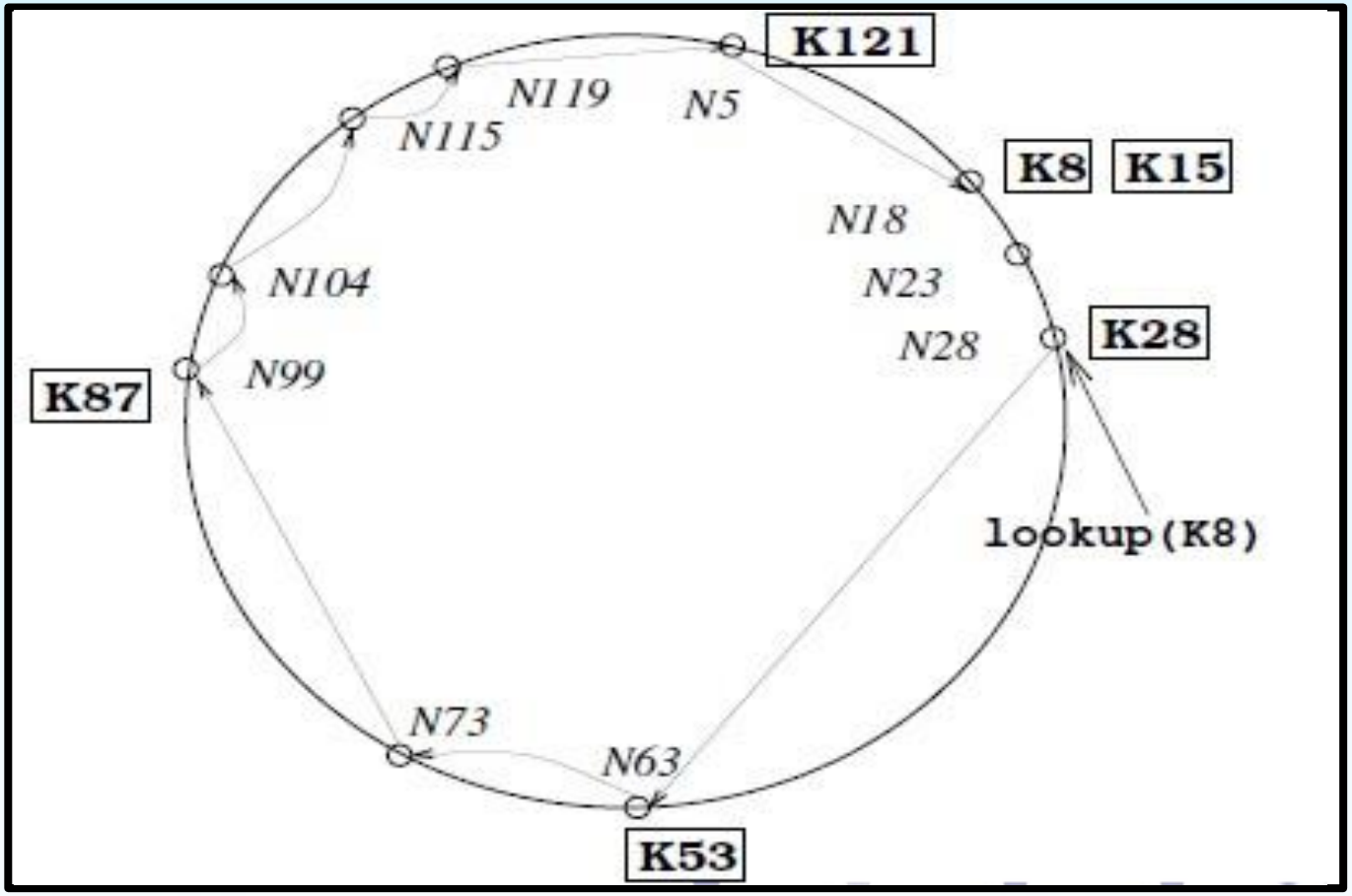
- **Mappings from node address space and object space** in a simple DHT. Highly deterministic placement of files/data allows fast lookup. But file insertions/deletions under churn incurs some cost.
- Attribute search, range search, keyword search etc. not possible.

Chord Distributed Hash Table: Overview

- The **Chord protocol, proposed by Stoica et al. (2003),** uses a flat key space to associate the mapping between network nodes and data objects / files /values.
- The node address as well as object value is mapped to a logical identifier in a common flat key space using a consistent hash function.
- **When a node joins or leaves the network of *n* nodes, only O(1*/n)* keys have to be moved from one location to another.**
- Two steps involved:
    1. Map the object value to its key
    2. Map the key to the node in the native address space using *lookup*

- **Common address space is a *m*-bit identifier (2^m addresses),** and this space is arranged on a **logical ring *mod* (2^m).**
- A **key *k*** gets assigned to the first node such that the node identifier equals or is greater than the key identifier of *k* in the common identifier space. The node is the **successor of k, denoted *succ(k).***

Example



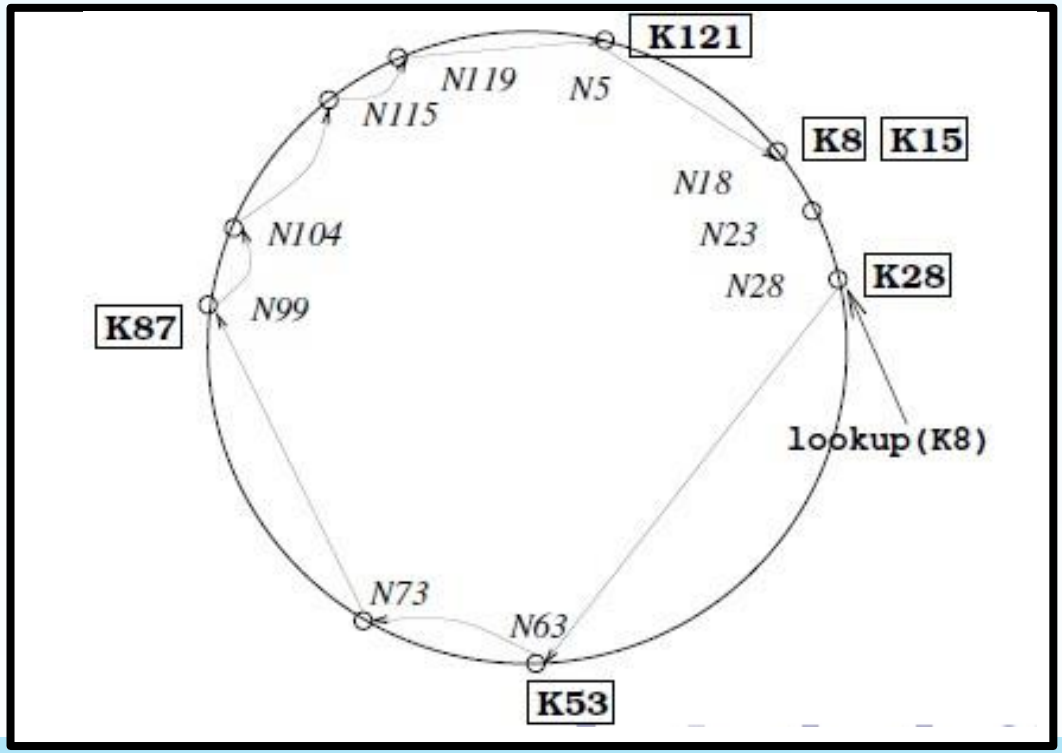
A Chord ring for m = 7 is depicted in figure. Nodes N5, N18, N23, N28, N63, N73, N99, N104, N115, and N119 are shown. Six keys, K8, K15, K28, K53, K87, and K121, are stored among these nodes using succ(k) as follows:
succ8 = 18, succ15 = 18, succ28 = 28, succ53 = 63, succ87 = 99, and succ121 = 5.

A **key *k*** gets assigned to the first node such that the node identifier equals or is greater than the key identifier of *k* in the common identifier space. The node is the **successor of k, denoted *succ(k).***

Chord: *Simple* Lookup

- Each node tracks its successor on the ring.
- Query for key x is forwarded on the ring until **reaches first node whose identifier y ≥ key (x mod (2^m))**.
- The result, **node with key y**, is returned to the querying node *(along the reverse of the path that was followed by the query.)*
- This mechanism **requires O(1) local space but *O(n)* hops.**

(variables)
integer: *successor* ← initial value;
(1) *i.Locate_Successor (key), where key f=i:*
(1a) **if** *key* ∈ (*i, successor*] **then**
(1b) **return**(*successor*)
(1c) **else return** *successor.Locate _Successor*(*key*).



Chord: *Scalable* Lookup

- Each **node *i*** maintains a routing table, called the ***finger table*, with *O*(*log n*) entries**, such that the ***x* th entry (1 ≤ *x* ≤ *m*)** is the node identifier of the node ***succ* (*i* + 2^{x-1}).**
- denoted by $i.finger[x] = succ(i + 2^{x-1})$.
- This is the first node whose key is greater than the key of node *i* by **at least 2^{x-1} *mod* 2^m .**
- **Complexity: *O*(*log n*) message hops** at the cost of ***O*(*log n*) space** in the local routing tables
- Due to the ***log* structure** of the finger table, there is more info about nodes closer by than about nodes further away.

Contd…

Consider a query on key ***key*** at node *i* ,

1. if ***key*** lies between node ***i*** and *its successor*, the ***key*** would reside at the successor and its address is returned.

2. If ***key*** lies beyond the successor, then **node *i*** searches through the ***m* entries** in its finger table to identify the node ***j*** such that *j* most immediately precedes ***key*** , among all the entries in the finger table.

3. As node *j* is the closest known node that precedes ***key, j*** is most likely to have the most information on locating *key* , i.e., locating the immediate successor node to which ***key*** has been mapped.

Chord: Scalable Lookup Procedure

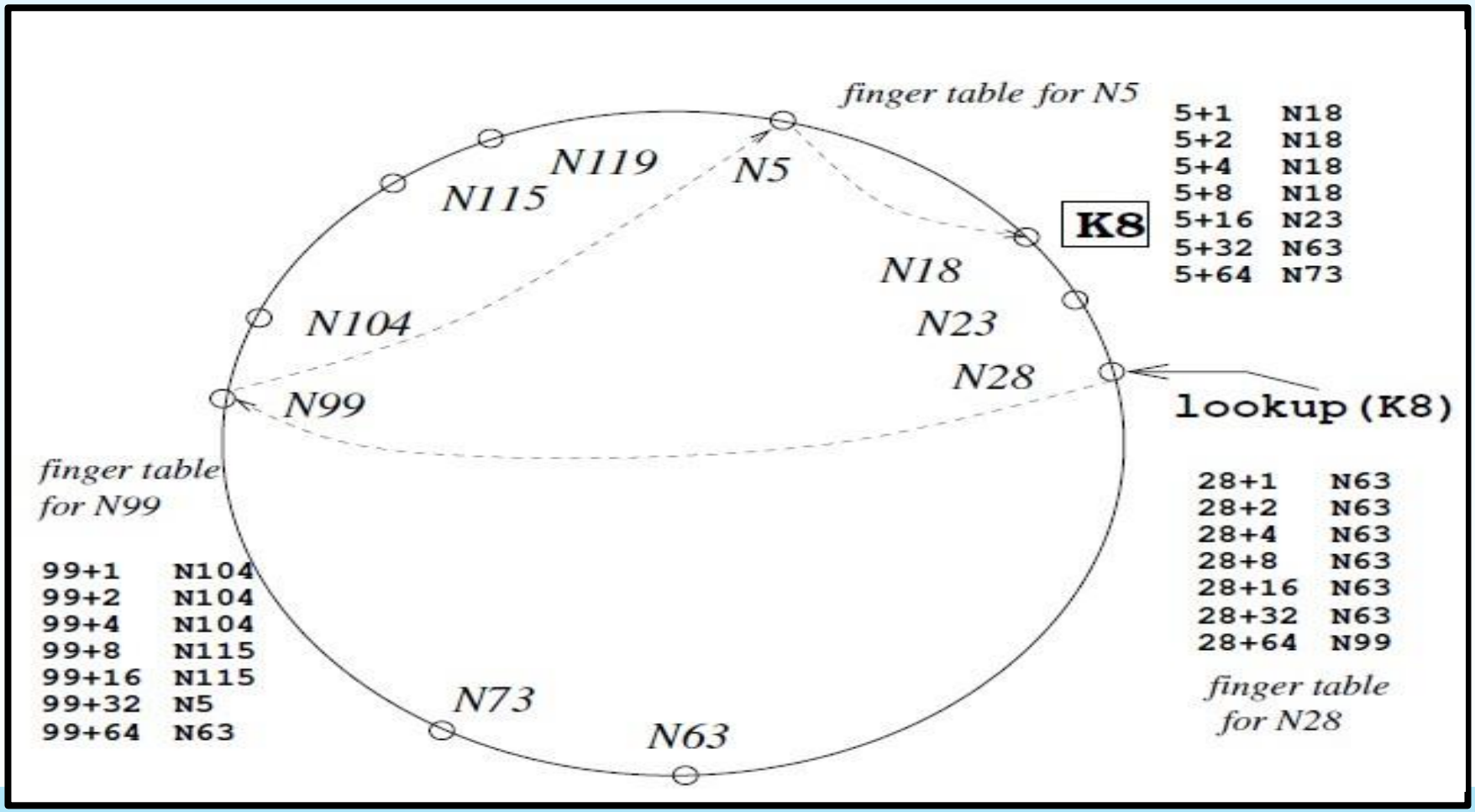
```
(variables)
integer: successor ← initial value;
integer: predecessor ← initial value;
array of integer finger [1 . . . log n];

(1)i.Locate Successor (key ), where key =/i:
(1a) if key ∈ (i, successor ] then
(1b)   return(successor )
(1c) else
(1d)       j ← Closest _Preceding _Node(key );
(1e) return j.Locate _Successor (key ).
(2)i.Closest_Preceding_Node(key ), where key = i:
(2a) for count =  m down  to 1 do
(2b)   if finger [count] ∈(i, key ] then
(2c)            break();
(2d) return(finger [count]).
```

Algorithm 2: **A scalable object location algorithm in Chord at node i.**

Chord: Scalable Lookup Example

Example: The use of the finger tables in answering the **query lookup(K8)**at **node N28** is illustrated in figure. The finger tables of **N28, N99, and N5** that are used are shown.



Managing Churn: Node joins

The code to manage dynamic node joins, departures, and failures is given in the **Algorithm 3** of managing churn

Node joins:

- To create a new ring, a node *i* executes ***Create_New_Ring*** which creates a ring with the singleton node. To join a ring that contains some node *j*, node *i* invokes ***Join_Ring(j)***.

- Node *j* locates *i's* successor on the logical ring and informs *i* of its successor. Before *i* can participate in the P2P exchanges, several actions need to happen: *i's* successor needs to update its predecessor entry to *i*, *i's* predecessor needs to revise its successor field to *i, i* needs to identify its predecessor, the finger table at *i* needs to be built, and the finger table of all nodes need to be updated to account for *i's* presence.

- This is achieved by procedures ***Stabilize(), Fix_Fingers()***, and ***Check_Predecessor()*** that are periodically invoked by each node.

Contd…

The Given Figure: Illustrates the main steps of the joining process. A recent joiner node *i* that has executed *Join_Ring(·)* gets integrated into the ring by the following sequence:

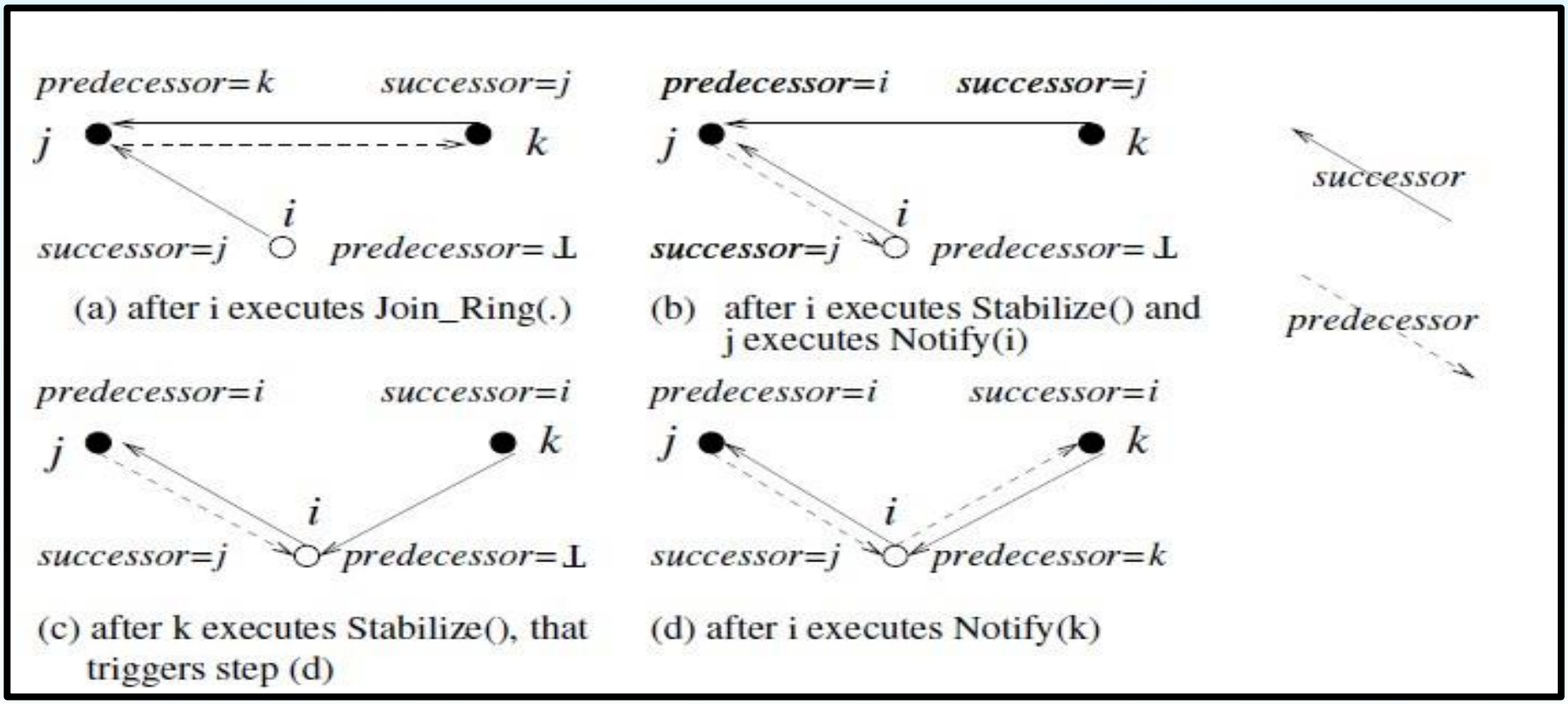


Figure: Steps in the integration of node *i* in the ring, where *j* > *i* > *k*

Contd…

- Once all the successor variables and finger tables have stabilized, a call by any node to ***Locate_Successor(·)*** will reflect the **new *joiner i.***
- Until then, a call to ***Locate_Successor(·)*** may result in the ***Locate_Successor(·)*** call performing a conservative scan.
- The loop in ***Closest_Preceding_Node*** that scans the finger table will result in a search traversal using smaller hops rather than truly logarithmic hops, resulting in some inefficiency.
- Still, the **node *i*** will be located although via more hops.

Managing Churn: Node failures and departures

Node failures and departures:

- When a node *j* fails abruptly, its successor *i* on the ring will discover the failure when the successor *i* executes ***Check_Predecessor*** **()** periodically.
- Process *i* gets a chance to update its *predecessor* field when another node *k* causes *i* to execute ***Notify(k)***. But that can happen only if *k's* ***successor*** variable is *i.*
- This requires the predecessor of the failed node to recognize that its successor has failed, and get a new functioning successor.
- In fact, the successor pointers are required for object search; the predecessor variables are required only to accommodate new joiners.
- Note from **Algorithm 3** that knowing that the successor is functional, and that the nodes pointed to by the finger pointers are functional, is essential.

Algorithm 3: Managing churn in Chord. Code shown is for node *i*

(variables)
integer: *successor* ← initial value;
integer: *predecessor* ← initial value;
array of integer *finger* [1 . . . *log m*];
integer: *next finger* ←1;
(1)i.Create_New_Ring ():
(1a) *predecessor* ←⊥;
(1b) *successor* ← *i* .
(2) i.Join_Ring (*j*), where *j* is any node on the ring to be joined:
(2a) *predecessor* ←⊥;
(2b) *successor* ←*j.Locate_Successor* (*i*).
(3) i.*Stabilize*(): // executed periodically to verify and inform successor
(3a) *x* ← *successor.predecessor* ;
(3b) **if** *x* ∈ (*i, successor*) **then**
(3c) *successor* ← *x* ;
(3d) *successor.Notify* (*i*).

Contd…

(4)i.Notify (*j*): // *j* believes it is predecessor of *i*

(4a) **if** *predecessor* = ⊥ **or** *j* ∈ (*predecessor* , *i*)) **then**

(4b) transfer keys in the range [*j*, i] to *j*;

(4c) *predecessor* ← *j*.

(5) i.Fix_Fingers():// executed periodically to update the finger table

5a) *next_finger* ← *next_finger* + 1;

(5b) **if** *next_finger* > *m* **then**

(5c) *next_finger* ← 1;

(5d) *finger* [*next_finger*] ← *Locate_Successor* (*i* + $2^{next\_finger-1}$).

(6)i.Check_Predecessor (): // executed periodically to verify whether //predecessor still exists

(6a) **if** predecessor has failed **then**

(6b) *predecessor* ← ⊥.

Complexity

- For a Chord network with ***n* nodes,** each node is responsible for **at most $(1 + \epsilon)K/n$ keys** with "high probability", where K is the total number of keys.
- Using consistent hashing, ϵ can be shown to be bounded by ***O*(*logn*).**
- The search for a successor ***Locate_Successor*** in a Chord network with ***n* nodes** requires **time complexity *O*(*log n*)** with high probability.
- The **size of the finger table is *log* (*n*) ≤ *m***
- The **average lookup time is 1/2 *log* (*n*).**

Comparison of Structure P2P Overlay Network (1)

Algorithm	Overlay Network Topology	Distance from i to j	Routing path length
Chord	One-dimensional ring	$(j - i) \bmod 2^m$	O(logN)
CAN	*d*-dimensional cube	Euclidean distance	$(d/4)\ N^{1/d}$
GISP	Structureless	objects: the difference of the two IDs; nodes: $(i,j)/(2^{st-1}2^{sj-1})$; si, sj are "peer strength"	uncertain
Kademila	XOR-tree	$i \oplus j$	$O(\log_{2b} N)$
Pastry	Tree + ring	assessed digit by digit	$O(\log_{2b} N)$
Tapestry	Tree	same as Pastry	$O(\log_{2b} N)$
Viceroy	Butterfly	$(j - i) \bmod 1$	O(logN)

Source: "A Survey on Distributed Hash Table (DHT): Theory, Platforms, and Applications", Hao Zhang, Yonggang Wen, Haiyong Xie, and Nenghai Yu., 2013

Comparison of Structure P2P Overlay Network(2)

Algorithm	Routing path selection	# of maintained peers	Node's joining	Node's leaving
Chord	Greedy Algorithm	O(log2 N)	Find successor; construct finger table	Run stabilization protocol periodically
CAN		2d	Generate neighbor list	Update neighbor lists
GISP		as many as possible	Generate routing list	Delete failing nodes from routing list
Kademila		at most 160 × k(O(log2 N))	Constructs k- buckets	Detect the target node before routing
Pastry		O(log2b N)	Generate routing table, neighbor- hood set and a namespace set	Detect the target node before routing
Tapestry		O(log2b N)	Construct the routing table	Heartbreak message
Viceroy		at most 7	Construct the three kinds of links	Repaired by the LOOKUP operation

Conclusion

- Peer-to-peer (P2P) networks allow equal participation and resource sharing among the users.
- This lecture first give the basic fundamentals and underlying concepts of P2P networks. *i.e.* **(i) Structured P2P networks and (ii) Unstructured P2P networks**
- Then we have discussed the concept of '**Distributed Hash Table'** and DHT based classical structured P2P network i.e. **Chord**