Chapter Outline

- Searching and addressing
  - Structured and unstructured networks

- Distributed Hash Tables
  - What they are?
  - How they work?
  - What are they good for?
  - Examples: Chord, CAN, Plaxton/Pastry/Tapestry

- Networks and graphs
  - Graph theory meets networking
  - Different types of graphs and their properties
Searching and Addressing

- Two basic ways to find objects:
  1. Search for them
  2. Address them using their unique name
- Both have pros and cons (see below)
- Most existing P2P networks built on searching, but some networks are based on addressing objects
- Difference between searching and addressing is a very fundamental difference
  - Determines how network is constructed
  - Determines how objects are placed
  - “Determines” efficiency of object location
- Let's compare searching and addressing
Addressing vs. Searching

- “Addressing” networks find objects by addressing them with their unique name (cf. URLs in Web)
- “Searching” networks find objects by searching with keywords that match objects’s description (cf. Google)

**Addressing**

- **Pros:**
  - Each object uniquely identifiable
  - Object location can be made efficient

- **Cons:**
  - Need to know unique name
  - Need to maintain structure required by addresses

**Searching**

- **Pros:**
  - No need to know unique names
  - More user friendly

- **Cons:**
  - Hard to make efficient
  - Can solve with money, see Google
  - Need to compare actual objects to know if they are same
### Addressing vs. Searching: Examples

<table>
<thead>
<tr>
<th></th>
<th>Searching</th>
<th>Addressing</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Physical name of object</strong></td>
<td>Searching in P2P networks,</td>
<td>URLs in Web</td>
</tr>
<tr>
<td></td>
<td>Searching in filesystem</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Desktop searches)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>(Search components of URL with Google?)</td>
<td></td>
</tr>
<tr>
<td><strong>Logical name of object</strong></td>
<td>?</td>
<td>Object names in DHT,</td>
</tr>
<tr>
<td></td>
<td>(Search components of URNs)</td>
<td>URNs</td>
</tr>
<tr>
<td><strong>Content or metadata of object</strong></td>
<td>Searching in P2P networks,</td>
<td>N/A</td>
</tr>
<tr>
<td></td>
<td>Standard Google search</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Desktop searches</td>
<td></td>
</tr>
</tbody>
</table>
Searching, Addressing, and P2P

- We can distinguish two main P2P network types
  - Unstructured networks/systems
    - Based on searching
    - Unstructured does **NOT** mean complete lack of structure
      - Network has graph structure, e.g., scale-free
    - Network has structure, but peers are free to join anywhere and objects can be stored anywhere
  - So far we have seen unstructured networks
  - Structured networks/systems
    - Based on addressing
    - Network structure determines where peers belong in the network and where objects are stored
    - How to build structured networks?
Another Classification of P2P Systems

- Sometimes P2P systems classified in generations
- No 100% consensus on what is in which generation
- 1st generation
  - Typically: Napster
- 2nd generation
  - Typically: Gnutella
- 3rd generation
  - Typically: Superpeer networks
- 4th generation
  - Typically: Distributed hash tables
  - Note: For DHTs, no division into generations yet
Distributed Hash Tables

- What are they?
- How they work?
- What are they good for?
- Examples:
  - Chord
  - CAN
  - Plaxton/Pastry/Tapestry
DHT: Motivation

- Why we need DHTs?
- Searching in P2P networks is not efficient
  - Either centralized system with all its problems
  - Or distributed system with all its problems
  - Hybrid systems cannot guarantee discovery either
- Actual file transfer process in P2P network is scalable
  - File transfers directly between peers
- Searching does not scale in same way
- Original motivation for DHTs: More efficient searching and object location in P2P networks
- Put another way: Use addressing instead of searching
Recall: Hash Tables

- Hash tables are a well-known data structure
- Hash tables allow insertions, deletions, and finds in constant (average) time
- Hash table is a fixed-size array
  - Elements of array also called hash buckets
- Hash function maps keys to elements in the array
- Properties of good hash functions:
  - Fast to compute
  - Good distribution of keys into hash table
  - Example: SHA-1 algorithm
Hash Tables: Example

- Hash function:
  \[ \text{hash}(x) = x \mod 10 \]
- Insert numbers 0, 1, 4, 9, 16, and 25
- Easy to find if a given key is present in the table
Distributed Hash Table: Idea

- Hash tables are fast for lookups
- Idea: Distribute hash buckets to peers
- Result is **Distributed Hash Table** (DHT)
- Need efficient mechanism for finding which peer is responsible for which bucket and routing between them
DHT: Principle

- In a DHT, each node is responsible for one or more hash buckets
  - As nodes join and leave, the responsibilities change
- Nodes communicate among themselves to find the responsible node
  - Scalable communications make DHTs efficient
- DHTs support all the normal hash table operations
Summary of DHT Principles

- Hash buckets distributed over nodes
- Nodes form an *overlay network*
  - Route messages in overlay to find responsible node
- Routing scheme in the overlay network is the difference between different DHTs
- DHT behavior and usage:
  - Node knows “object” name and wants to find it
    - Unique and known object names assumed
  - Node routes a message in overlay to the responsible node
  - Responsible node replies with “object”
    - Semantics of “object” are application defined
DHT Examples

- In the following look at some example DHTs
  - Chord
  - CAN
  - Tapestry
- Several others exist too
  - Pastry, Plaxton, Kademlia, Koorde, Symphony, P-Grid, CARP, …

- All DHTs provide the same abstraction:
  - DHT stores key-value pairs
  - When given a key, DHT can retrieve/store the value
  - No semantics associated with key or value
- Difference is in overlay routing scheme
Chord

- Chord was developed at MIT
- Originally published in 2001 at Sigcomm conference

- Chord’s overlay routing principle quite easy to understand
  - Paper has mathematical proofs of correctness and performance

- Many projects at MIT around Chord
  - CFS storage system
  - Ivy storage system
  - Plus many others…
Chord: Basics

- Chord uses SHA-1 hash function
  - Results in a 160-bit object/node identifier
  - Same hash function for objects and nodes
- Node ID hashed from IP address
- Object ID hashed from object name
  - Object names somehow assumed to be known by everyone
- SHA-1 gives a 160-bit identifier space
- Organized in a ring which wraps around
  - Nodes keep track of predecessor and successor
  - Node responsible for objects between its predecessor and itself
  - Overlay is often called “Chord ring” or “Chord circle”
Chord: Examples

Below examples for:
- How to join the Chord ring
- How to store and retrieve values
Joining: Step-By-Step Example

- Setup: Existing network with nodes on 0, 1 and 4

- Note: Protocol messages simply examples

- Many different ways to implement Chord
  - Here only conceptual example
  - Covers all important aspects
Joining: Step-By-Step Example: Start

- New node wants to join
- Hash of the new node: 6
- Known node in network: Node1
- Contact Node1
  - Include own hash
Joining: Step-By-Step Example: Situation Before Join

- **pred0**: No data
- **succ0**: Data for [4;0]
- **pred1**: No data
- **succ1**: Data for [0;1]
- **pred4**: No data
- **succ4**: Data for [1;4]
Joining: Step-By-Step Example:
Contact known node

- Arrows indicate open connections
- Example assumes connections are kept open, i.e., messages processed recursively
- Iterative processing is also possible
Joining: Step-By-Step Example:
Join gets routed along the network
Joining: Step-By-Step Example: Successor of New Node Found
Joining: Step-By-Step Example: Joining Successful + Transfer

Joining is successful

Old responsible node transfers data that should be in new node

New node informs Node4 about new successor (not shown)

Note: Transferring can happen also later
Joining: Step-By-Step Example:
All Is Done
Storing a Value

- Node 6 wants to store an object with name “Foo” and value 5
- hash(Foo) = 2
Storing a Value

STORE 2 5
Storing a Value

STORE 2 5
Storing a Value

Value is now stored in node 4.
Retrieving a Value

- Node 1 wants to get object with name “Foo”
- hash(Foo) = 2
  - Foo is stored on node 4
Retrieving a Value

Retriew 2
Retrieving a Value
Chord: Scalable Routing

- Routing happens by passing message to successor
- What happens when there are 1 million nodes?
  - On average, need to route 1/2-way across the ring
  - In other words, 0.5 million hops! Complexity $O(n)$
- How to make routing scalable?
  - **Answer:** Finger tables
- Basic Chord keeps track of predecessor and successor
- Finger tables keep track of more nodes
  - Allow for faster routing by jumping long way across the ring
  - Routing scales well, but need more state information
- Finger tables not needed for correctness, only performance improvement
Chord: Finger Tables

- In $m$-bit identifier space, node has up to $m$ fingers
- Fingers are stored in the finger table

- Row $i$ in finger table at node $n$ contains first node $s$ that succeeds $n$ by at least $2^{i-1}$ on the ring
- In other words:
  \[ \text{finger}[i] = \text{successor}(n + 2^{i-1}) \]
- First finger is the successor
- Distance to $\text{finger}[i]$ is at least $2^{i-1}$
Chord: Scalable Routing

- Finger intervals increase with distance from node $n$
  - If close, short hops and if far, long hops

Two key properties:
- Each node only stores information about a small number of nodes
- Cannot in general determine the successor of an arbitrary ID

Example has three nodes at 0, 1, and 4
- 3-bit ID space --> 3 rows of fingers
Chord: Performance

- Search performance of “pure” Chord $O(n)$
  - Number of nodes is $n$
- With finger tables, need $O(\log n)$ hops to find the correct node
  - Fingers separated by at least $2^{i-1}$
  - With high probability, distance to target halves at each step
  - In beginning, distance is at most $2^m$
  - Hence, we need at most $m$ hops
- For state information, “pure” Chord has only successor and predecessor, $O(1)$ state
- For finger tables, need $m$ entries
  - Actually, only $O(\log n)$ are distinct
  - Proof is in the paper
CAN: Content Addressable Network

- CAN developed at UC Berkeley
- Originally published in 2001 at Sigcomm conference(!)

- CANs overlay routing easy to understand
  - Paper concentrates more on performance evaluation
  - Also discussion on how to improve performance by tweaking

- CAN project did not have much of a follow-up
  - Only overlay was developed, no bigger follow-ups
CAN: Basics

- CAN based on N-dimensional Cartesian coordinate space
  - Our examples: $N = 2$
  - One hash function for each dimension
- Entire space is partitioned amongst all the nodes
  - Each node owns a zone in the overall space

Abstractions provided by CAN:
- Can store data at points in the space
- Can route from one point to another

Point = Node that owns the zone in which the point (coordinates) is located
Order in which nodes join is important
CAN: Partitioning
CAN: Partitioning

1

2
CAN: Partitioning
CAN: Partitioning
CAN forms a d-dimensional torus
CAN: Examples

Below examples for:

- How to join the network
- How routing tables are managed
- How to store and retrieve values
Discover some node “I” already in CAN
CAN: Node Insertion

New node picks its coordinates in space

New node picks its coordinates in space

(p,q) - pick random point in space
I routes to (p,q), and discovers that node J owns (p,q)
CAN: Node Insertion

Split J’s zone in half. New owns one half
CAN: Routing Table
CAN: Routing

Diagram showing a grid with two points (x, y) and (a, b) connected by a path.
node I::insert(K,V)

\[ a = h_x(K) \]
\[ b = h_y(K) \]
node I::insert(K,V)

(1) \( a = h_x(K) \)
   \( b = h_y(K) \)

(2) route(K,V) -\(\rightarrow\) (a,b)
node \texttt{l::insert(K,V)}

1. \(a = h_x(K)\)
   \(b = h_y(K)\)

2. \texttt{route(K,V) -> (a,b)}

3. \((a,b)\) stores \((K,V)\)
node J::retrieve(K)

(1) \( a = h_x(K) \)
\( b = h_y(K) \)

(2) route "retrieve(K)" to (a, b)
### CAN: Improvements

- Possible to increase number of dimensions $d$
  - Small increase in routing table size
  - Shorter routing path, more neighbors for fault tolerance
- Multiple realities (= coordinate spaces)
  - Use more hash functions
  - Same properties as increased dimensions
- Routing weighted by round-trip times
  - Take into account network topology
  - Forward to the “best” neighbor
CAN: More Improvements

- Use well-known landmark servers (e.g., DNS roots)
  - Nodes join CAN in different areas, depending on distance to landmarks
    - Pick points “near” landmark
  - Idea: Geographically close nodes see same landmarks

- Uniform partitioning
  - New node splits the largest zone in the neighborhood instead of the zone of the responsible node
CAN: Performance

- State information at node $O(d)$
  - Number of dimensions is $d$
  - Need two neighbors in all coordinate axis
  - Independent of the number of nodes!

- Routing takes $O(dn^{1/d})$ hops
  - Network has $n$ nodes
  - Multiple dimensions and realities improve this
  - For routing: multiple dimensions are better
  - But: multiple realities improve availability and fault tolerance
Tapestry

- Tapestry developed at UC Berkeley(!)
  - Different group from CAN developers
- Tapestry developed in 2000, but published in 2004
  - Originally only as technical report, 2004 as journal article
- Many follow-up projects on Tapestry
  - Example: OceanStore

- Tapestry based on work by Plaxton et al.
- Plaxton network has also been used by Pastry
- Pastry was developed at Microsoft Research and Rice University
  - Difference between Pastry and Tapestry minimal
  - Tapestry and Pastry add dynamics and fault tolerance to Plaxton network
Tapestry: Plaxton Network

- Plaxton network (or Plaxton mesh) based on prefix routing (similar to IP address allocation)
  - Prefix and postfix are functionally identical
  - Tapestry originally postfix, now prefix?!
- Node ID and object ID hashed with SHA-1
  - Expressed as hexadecimal (base 16) numbers (40 digits)
  - Base is very important, here we use base 16

- Each node has a neighbor map with multiple levels
  - Each level represents a matching prefix up to digit position in ID
  - A given level has number of entries equal to the base of ID
  - \(i\)th entry in \(j\)th level is closest node which starts \(\text{prefix}(N,j-1)+i\)
  - Example: 9th entry of 4th level for node 325AE is the closest node with ID beginning with 3259
Tapestry: Routing Mesh

- (Partial) routing mesh for a single node 4227
- Neighbors on higher levels match more digits
### Tapestry: Neighbor Map for 4227

<table>
<thead>
<tr>
<th>Level</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>8</th>
<th>A</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1D76</td>
<td>27AB</td>
<td></td>
<td>51E5</td>
<td>6F43</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>43C9</td>
<td>44AF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>42A2</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>4228</td>
<td></td>
</tr>
</tbody>
</table>

- There are actually 16 columns in the map (base 16)
- Normally more (most?) entries would be filled
- Tapestry has neighbor maps of size 40 x 16
Tapestry: Routing Example

- Route message from 5230 to 42AD
- Always route to node closer to target
  - At $n^{th}$ hop, look at $(n+1)^{th}$ level in neighbor map --> “always” one digit more
- Not all nodes and links are shown
Tapestry: Properties

- Node responsible for objects which have same ID
  - Unlikely to find such node for every object
  - Node responsible also for “nearby” objects (surrogate routing, see below)

Object publishing:
- Responsible nodes store only pointers
  - Multiple copies of object possible
  - Each copy must publish itself
- Pointers cached along the publish path
- Queries routed towards responsible node
- Queries “often” hit cached pointers
  - Queries for same object go (soon) to same nodes

Note: Tapestry focuses on storing objects
- Chord and CAN focus on values, but in practice no difference
Tapestry: Publishing Example

- Two copies of object “DOC” with ID 4377 created at AA93 and 4228
- AA93 and 4228 publish object DOC, messages routed to 4377
  - Publish messages create location pointers on the way
- Any subsequent query can use location pointers
Tapestry: Querying Example

- Requests initially route towards 4377
- When they encounter the publish path, use location pointers to find object
- Often, no need to go to responsible node
- Downside: Must keep location pointers up-to-date
Tapestry: Making It Work

- Previous examples show a Plaxton network
  - Requires global knowledge at creation time
  - No fault tolerance, no dynamics
- Tapestry adds fault tolerance and dynamics
  - Nodes join and leave the network
  - Nodes may crash
  - Global knowledge is impossible to achieve
- Tapestry picks closest nodes for neighbor table
  - Closest in IP network sense (= shortest RTT)
  - Network distance (usually) transitive
    - If A is close to B, then B is also close to A
  - Idea: Gives best performance
Tapestry: Fault-Tolerant Routing

- Tapestry keeps mesh connected with keep-alives
  - Both TCP timeouts and UDP “hello” messages
  - Requires extra state information at each node

- Neighbor table has backup neighbors
  - For each entry, Tapestry keeps 2 backup neighbors
  - If primary fails, use secondary
    - Works well for uncorrelated failures

- When node notices a failed node, it marks it as invalid
  - Most link/connection failures short-lived
  - Second chance period (e.g., day) during which failed node can come back and old route is valid again
  - If node does not come back, one backup neighbor is promoted and a new backup is chosen
Tapestry: Fault-Tolerant Location

- Responsible node is a single point of failure
- **Solution:** Assign multiple roots per object
  - Add “salt” to object name and hash as usual
  - Salt = globally constant sequence of values (e.g., 1, 2, 3, …)
- Same idea as CAN’s multiple realities
- This process makes data more available, even if the network is partitioned
  - With s roots, availability is $P \approx 1 - (1/2)^s$
  - Depends on partition
- These two mechanisms “guarantee” fault-tolerance
  - In most cases :-)
  - Problem: If the only out-going link fails…
Tapestry: Surrogate Routing

- Responsible node is node with same ID as object
  - Such a node is unlikely to exist
- Solution: surrogate routing
- What happens when there is no matching entry in neighbor map for forwarding a message?
- Node picks (deterministically) one entry in neighbor map
  - Details are not explained in the paper :(
- Idea: If “missing links” are deterministically picked, any message for that ID will end up at same node
  - This node is the surrogate
- If new nodes join, surrogate may change
  - New node is neighbor of surrogate
Tapestry: Performance

- Messages routed in $O(\log_b N)$ hops
  - At each step, we resolve one more digit in ID
  - $N$ is the size of the namespace (e.g., SHA-1 = 40 digits)
  - Surrogate routing adds a bit to this, but not significantly

- State required at a node is $O(b \log_b N)$
  - Tapestry has $c$ backup links per neighbor, $O(cb \log_b N)$
  - Additionally, same number of backpointers
## DHT: Comparison

<table>
<thead>
<tr>
<th></th>
<th>Chord</th>
<th>CAN</th>
<th>Tapestry</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type of network</td>
<td>Ring</td>
<td>N-dimensional</td>
<td>Prefix routing</td>
</tr>
<tr>
<td>Routing</td>
<td>$O(\log n)$</td>
<td>$O(d \cdot n^{1/d})$</td>
<td>$O(\log_b N)$</td>
</tr>
<tr>
<td>State</td>
<td>$O(\log n)$</td>
<td>$O(d)$</td>
<td>$O(b \cdot \log_b N)$</td>
</tr>
<tr>
<td>Caching efficient</td>
<td>++</td>
<td>++</td>
<td>++</td>
</tr>
<tr>
<td>Robustness</td>
<td>+/-</td>
<td>+++</td>
<td>++</td>
</tr>
<tr>
<td>IP Topology-Aware</td>
<td>N</td>
<td>N/Y</td>
<td>Y</td>
</tr>
<tr>
<td>Used for other projects</td>
<td>+++</td>
<td>--</td>
<td>++</td>
</tr>
</tbody>
</table>

**Note:** $n$ is number of nodes, $N$ is size of Tapestry’s namespace
Other DHTs

- Many other DHTs exist too
  - Pastry, similar to Tapestry
  - Kademlia, uses XOR metric
  - Kelips, group nodes into $k$ groups, similar to KaZaA
  - Plus some others…
- Overnet P2P network (also eDonkey) uses Kademlia
  - Wide-spread deployed DHT

- All DHTs provide same API
  - In principle, DHT-layer is interchangeable
Networks and Graphs

- Refresher of graph theory
- Graph families and models
  - Random graphs
  - Small world graphs
  - Scale-free graphs
- Graph theory and P2P
  - How are the graph properties reflected in real systems?
What Is a Graph?

Definition of a graph:
Graph $G = (V, E)$ consists of two finite sets, set $V$ of vertices (nodes) and set $E$ of edges (arcs) for which the following applies:

1. If $e \in E$, then exists $(v, u) \in V \times V$, such that $v \in e$ and $u \in e$
2. If $e \in E$ and above $(v, u)$ exists, and further for $(x, y) \in V \times V$ applies $x \in e$ and $y \in e$, then $\{v, u\} = \{x, y\}$

Example graph with 4 vertices and 5 edges
Properties of Graphs

- An edge $e \in E$ is **directed** if the start and end vertices in condition 2 above are identical: $v = x$ and $y = u$
- An edge $e \in E$ is **undirected** if $v = x$ and $y = u$ as well as $v = y$ and $u = x$ are possible
- A graph $G$ is **directed** (undirected) if the above property holds for all edges
- A **loop** is an edge with identical endpoints
- Graph $G_1 = (V_1, E_1)$ is a **subgraph** of $G = (V, E)$, if $V_1 \subseteq V$ and $E_1 \subseteq E$ (such that conditions 1 and 2 are met)
Important Types of Graphs

- Vertices $v, u \in V$ are connected if there is a path from $v$ to $u$: $(v, v_2), (v_2, v_3), \ldots, (v_{k-1}, u) \in E$
- Graph $G$ is connected if all $v, u \in V$ are connected
- Undirected, connected, acyclic graph is called a tree
  - Sidenote: Undirected, acyclic graph which is not connected is called a forest
- Directed, connected, acyclic graph is also called DAG
  - DAG = directed, acyclic graph (connected is “assumed”)
- An induced graph $G(V_C) = (V_C, E_C)$ is a graph $V_C \subseteq V$ and with edges $E_C = \{e = (i, j) \mid i, j \in V_C\}$
- An induced graph is a component if it is connected
Vertex Degree

- In graph $G = (V, E)$, the degree of vertex $v \in V$ is the total number of edges $(v, u) \in E$ and $(u, v) \in E$
  - Degree is the number of edges which touch a vertex
- For directed graph, we distinguish between in-degree and out-degree
  - In-degree is number of edges coming to a vertex
  - Out-degree is number of edges going away from a vertex
- The degree of a vertex can be obtained as:
  - Sum of the elements in its row in the incidence matrix
  - Length of its vertex incidence list
Important Graph Metrics

- **Distance**: $d(v, u)$ between vertices $v$ and $u$ is the length of the shortest path between $v$ and $u$
- **Average path length**: Sum of the distances over all pairs of nodes divided by the number of pairs
- **Diameter**: $d(G)$ of graph $G$ is the maximum of $d(v, u)$ for all $v, u \in V$
Six Degrees of Separation

- Famous experiment from 1960’s (S. Milgram)
- Send a letter to random people in Kansas and Nebraska and ask people to forward letter to a person in Boston
  - Person identified by name, profession, and city
- Rule: Give letter only to people you know by first name and ask them to pass it on according to same rule
- Some letters reached their goal
- Letter needed six steps on average to reach the person
- Graph theoretically: Social networks have dense local structure, but (apparently) small diameter
- How to model such networks?
Random Graphs

Random graphs are first widely studied graph family
- Many P2P networks choose neighbors more or less randomly

Two different notations generally used:
- Erdös and Renyi
- Gilbert (we will use this)

Gilbert’s definition: Graph $G_{n,p}$ (with $n$ nodes) is a graph where the probability of an edge $e = (v, w)$ is $p$

Construction algorithm:
- For each possible edge, draw a random number
- If the number is smaller than $p$, then the edge exists
- $p$ can be function of $n$ or constant
Basic Results for Random Graphs

Giant Connected Component:

Let \( c > 0 \) be a constant and \( p = c/n \). If \( c < 1 \) every component of \( G_{n,p} \) has order \( O(\log N) \) with high probability. If \( c > 1 \) then there will be one component of size \( n^*(f(c) + O(1)) \) where \( f(c) > 0 \), with high probability. All other components have size \( O(\log N) \).

- In plain English: Giant connected component emerges with high probability when average degree is about 1.

Node degree distribution

- If we take random node, how high is probability \( P(k) \) that node has degree \( k \)?
- Node degree is Poisson distributed \( P(k) = \frac{c^k e^{-c}}{k!} \)
More Basic Results

Diameter of a random graph
- If $pn/\log(n) \to \infty$ and $\log(n)/\log(pn) \to \infty$ then the diameter of $G_{n,p}$ is asymptotic to $\log(n)/\log(pn)$ with high probability

Clustering coefficient
- Clustering coefficient measures number of edges between neighbors divided by maximum number of edges between them (clique-like)
- Clustering coefficient $C(i)$ is defined as $C(i) = \frac{E(N(i))}{d(i)(d(i) - 1)}$
  - $E(N(i)) = \text{number of edges between neighbors of } i$
  - $d(i) = \text{degree of } i$
- Clustering coefficient of a random graph is asymptotically equal to $p$ with high probability
Random Graphs: Summary

- Before random graphs, regular graphs were popular
  - Regular: Every node has same degree
- Random graphs have two advantages over regular graphs
  1. Many interesting properties analytically solvable
  2. Much better for applications, e.g., social networks
- Note: Does not mean social networks are random graphs; just that the properties of social networks are well-described by random graphs
- Question: How to model networks with local clusters and small diameter?
- Answer: Small-world networks
Small-World Networks

- Developed/discovered by Watts and Strogatz (1998)
  - Over 30 years after Milgram’s experiment!
- Watts and Strogatz looked at three networks
  - Film collaboration between actors
  - US power grid
  - Neural network of worm C. elegans
- Results:
  - Compared to a random graph with same number of nodes
  - Diameters similar, slightly higher for real graph
  - Clustering coefficient orders of magnitude higher

Definition of small-worlds network:
- Dense local clustering structure and small diameter comparable to that of a same-sized random graph
Constructing Small-World Graphs

- Put all $n$ nodes on a ring, number them consecutively from 1 to $n$
- Connect each node with its $k$ clockwise neighbors
- Traverse around ring in clockwise order
- For every edge:
  - Draw random number $r$
  - If $r < p$, then re-wire edge by selecting a random target node from the set of all nodes (no duplicates)
  - Otherwise keep old edge
- Different values of $p$ give different graphs
  - If $p$ is close to 0, then original structure mostly preserved
  - If $p$ is close to 1, then new graph is random
  - Interesting things happen when $p$ is somewhere in-between
Regular, Small-World, Random

Regular

Small-World

Random

p = 0

p = 1
Problems with Small-World Graphs

Small-world graphs explain why:

- Highly clustered graphs can have short average path lengths

Small-world graphs do *NOT* explain why:

- This property emerges in real networks
  - Real networks are practically never ring-like

Further problem with small-world graphs:

- Nearly all nodes have same degree
- Not true for random graphs \((k \text{ edges } \sim c^k/k!)\)
- Is same true for real networks too?
- Let’s look at the Internet…
Internet

- Famous study by Faloutsos et al. (3 brothers! ;-) in 1999
- They examined Internet topology during 1998
  - AS-level topology, during 1998 Internet grew 45%

Motivation for work:

- What does the Internet look like?
- Are there any topological properties that don’t change over time?
- How can I generate Internet-like graphs for simulations?
Faloutsos Results

- 4 key properties, each follows a power-law
- Sort nodes according to their (out)degree
  1. Outdegree of a node is proportional to its rank to the power of a constant
  2. Number of nodes with same outdegree is proportional to the outdegree to the power of a constant
  3. Eigenvalues of a graph are proportional to the order to the power of a constant
  4. Total number of pairs of nodes within a distance $d$ is proportional to $d$ to the power of a constant

• Why would Internet obey such laws?
Answer: Power-Law Networks

- Also known as scale-free networks
- Barabasi-Albert-Model
  1. Network grows in time
  2. New node has preferences to whom it wants to connect
- Preferential connectivity modeled as
  - Each new node wants to connect to $m$ other nodes
  - Probability that an existing node $j$ gets one of the $m$ connections is proportional to its degree $d(j)$
- New nodes tend to connect to well-connected nodes
- Another way to express this is “rich get richer”
Applications to Peer-to-Peer

- Small-world model explains why short paths exist
- Why can we find these paths?
  - Each node has only local information
  - Milgram’s results showed first steps were the largest
- How to model this?
- Kleinberg’s Small-World Model
  - Set of points in an $n \times n$ grid
  - Distance is the number of “steps” separating points
    - $d(i, j) = |x_i - x_j| + |y_i - y_j|$  
- Construct graph as follows:
  - Every node $i$ is connected to node $j$ within distance $q$
  - For every node $i$, additional $q$ edges are added. Probability that node $j$ is selected is proportional to $d(i, j)^r$, for some constant $r$
Navigation in Kleinberg’s Model

- We want to send a message to another node
- Algorithm is decentralized if sending node only knows:
  - Its local neighbors
  - Position of the target node on the grid
  - Locations and long-range contacts of all nodes who come in contact of the message (not needed below, actually)
- Can be shown: Number of messages needed is proportional to $O(\log n)$ (only one correct $r$ per case)
- Practical algorithm: Forward message to contact who is closest to target
- Note: Kleinberg’s model assumes some way of associating nodes with points in grid
  - Compare with CAN DHT in Chapter 3
Power Law Networks and P2P

- Robustness comparison of random and power-law graphs
- Take network of 10000 nodes (random and power-law) and remove nodes randomly

**Random graph:**
- Take out 5% of nodes: Biggest component 9000 nodes
- Take out 18% of nodes: No biggest component, all components between 1 and 100 nodes
- Take out 45% of nodes: Only groups of 1 or 2 survive

**Power-law graph:**
- Take out 5% of nodes: Only isolated nodes break off
- Take out 18% of nodes: Biggest component 8000 nodes
- Take out 45% of nodes: Large cluster persists, fragments small

Recall Gnutella: Applies **ONLY for random failures**
Summary of Graphs

- Three kinds of graph models:
  - Random graph
  - Small-World
  - Power-Law (Scale-Free)
- Small-world graphs explain why we can have high clustering and short average paths
- Power-law graphs explain how graphs are built in many real networks
Chapter Summary

- Searching and addressing
  - Fundamental difference
  - Unstructured vs. structured networks
- Distributed Hash Tables
  - DHT provides a key to value mapping
  - Three examples: Chord, CAN, Tapestry
- Different networks and graphs
  - Random, small world, scale-free networks