Chapter 18: Peer-to-peer Computing and Overlay Graphs

Ajay Kshemkalyani and Mukesh Singhal

Distributed Computing: Principles, Algorithms, and Systems

Cambridge University Press

Characteristics

- P2P network: application-level organization of the network to flexibly share resources
- All nodes are equal; communication directly between peers (no client-server)
- Allow location of arbitrary objects; no DNS servers required
- Large combined storage, CPU power, other resources, without scalability costs
- Dynamic insertion and deletion of nodes, as well as of resources, at low cost

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: Desirable characteristics and performance features of P2P systems.

Napster

Central server maintains a table with the following information of each registered client: (i) the client's address (IP) and port, and offered bandwidth, and (ii) information about the files that the client can allow to share.

- A client connects to a meta-server that assigns a lightly-loaded server.
- The client connects to the assigned server and forwards its query and identity.
- The server responds to the client with information about the users connected to it and the files they are sharing.
- On receiving the response from the server, the client chooses one of the users from whom to download a desired file. The address to enable the P2P connection between the client and the selected user is provided by the server to the client.

Users are generally anonymous to each other. The directory serves to provide the mapping from a particular host that contains the required content, to the IP address needed to download from it.

Structured and Unstructured Overlays

- Search for data and placement of data depends on P2P overlay (which can be thought of as being below the application level overlay)
- Search is data-centric, not host-centric
- Structured P2P overlays:
  - E.g., hypercube, mesh, de Bruijn graphs
  - rigid organizational principles for object storage and object search
- Unstructured P2P overlays:
  - Loose guidelines for object search and storage
  - Search mechanisms are ad-hoc, variants of flooding and random walk
- Object storage and search strategies are intricately linked to the overlay structure as well as to the data organization mechanisms.

Data indexing

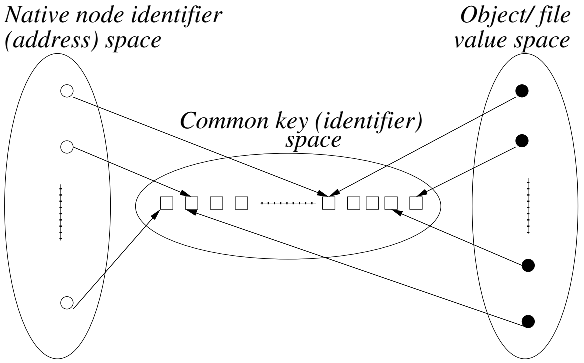
Data identified by indexing, which allows physical data independence from apps.

- 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.

Simple Distributed Hash Table scheme


Native node identifier (address) space
Object/ file value space
Common key (identifier) space

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.

Structured vs. unstructured overlays

Structured Overlays:

- structure $\Longrightarrow$ 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

Unstructured Overlays:

- No structure for overlay $\Longrightarrow$ 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
- Some overlay topologies naturally emerge:
  - Power Law Random Graph (PLRG) where node degrees follow a power law. Here, if the nodes are ranked in terms of degree, then the i^{th} node has c/i^{α} neighbors, where c is a constant.
  - simple random graph: nodes typically have a uniform degree

Unstructured Overlays: Properties

- Semantic indexing possible $\Longrightarrow$ keyword, range, attribute-based queries
- Easily accommodate high churn
- Efficient when data is replicated in network
- Good if user satisfied with "best-effort" search
- Network is not so large as to cause high delays in search

Gnutella features

- A joiner connects to some standard nodes from Gnutella directory
- *Ping* used to discover other hosts; allows new host to announce itself
- *Pong* in response to *Ping*; *Pong* contains IP, port #, max data size for download
- *Query* msgs used for flooding search; contains required parameters
- *QueryHit* are responses. If data is found, this message contains the IP, port #, file size, download rate, etc. Path used is reverse path of *Query*.

Search in Unstructured Overlays

Consider a system with n nodes and m objects. Let q_i be the popularity of object i, as measured by the fraction of all queries that are queries for object i. All objects may be equally popular, or more realistically, a Zipf-like power law distribution of popularity exists. Thus,

$$\sum_{i=1}^{m} q_i = 1 \tag{1}$$

$$\text{Uniform: } q_i = 1/m; \qquad \text{Zipf-like: } q_i \propto i^{-\alpha} \tag{2}$$

Let r_i be the number of replicas of object i, and let p_i be the fraction of all objects that are replicas of i. Three static replication strategies are: uniform, proportional, and square root. Thus,

$$\sum_{i=1}^{m} r_i = R; \qquad p_i = r_i/R \tag{3}$$

$$\text{Uniform: } r_i = R/m; \qquad \text{Proportional: } r_i \propto q_i; \qquad \text{Square-root: } r_i \propto \sqrt{q_i} \tag{4}$$

Under uniform replication, all objects have an equal number of replicas; hence the performance for all query rates is the same. With a uniform query rate, proportional and square-root replication schemes reduce to the uniform replication scheme.

Search in Unstructured Overlays

For an object search, some of the metrics of efficiency:

- probability of success of finding the queried object.
- delay or the number of hops in finding an object.
- the number of messages processed by each node in a search.
- node coverage, the fraction of (distinct) nodes visited
- *message duplication*, which is (#messages - #nodes visited)/#messages
- maximum number of messages at a node
- *recall*, the number of objects found satisfying the desired search criteria. This metric is useful for keyword, inexact, and range queries.
- *message efficiency*, which is the recall per message used

Guided versus Unguided Search. In unguided or blind search, there is no history of earlier searches. In guided search, nodes store some history of past searches to aid future searches. Various mechanisms for caching hints are used. We focus on unguided searches in the context of unstructured overlays.

Search in Unstructured Overlays: Flooding and Random Walk

- Flooding: with checking, with TTL or hop count, expanding ring strategy
- Random Walk: *k* random walkers, with checking
- Relationships of interest
  - The success rate as a function of the number of message hops, or TTL.
  - The number of messages as a function of the number of message hops, or TTL.
  - The above metrics as the replication ratio and the replication strategy changes.
  - The node coverage, recall, and message efficiency, as a function of the number of hops, or TTL; and of various replication ratios and replication strategies.
- Overall, *k*-random walk outperforms flooding

Replication Strategies

n	number of nodes in the system
m	number of objects in the system
q_i	normalized query rate, where $\sum_{i=1}^{m} q_i = 1$
r_i	number of replicas of object i
ρ	capacity (measured as number of objects) per node
R	$n\rho = \sum_{i=1}^{m} r_i$, the total capacity in the system
p_i	r_i/R, the population fraction of object i replicas
A_i	Average search size for i
A	Average search size for the system

Table: Parameters to study replication.

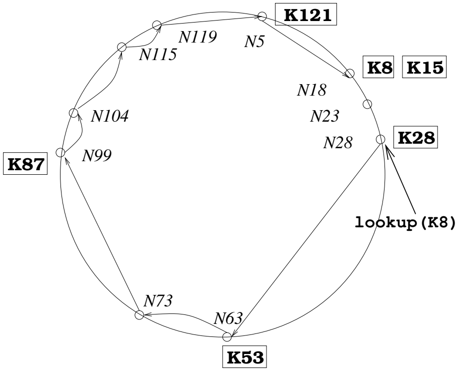
	r_i	A	$A_i = n/r_i$	$u_i = Rq_i/r_i$
Uniform	constant, R/m	m/ρ	m/ρ	$q_i m$
Proportional	$q_i R$	m/ρ	$1/(\rho q_i)$	1
Square-root	$R\sqrt{q_i}/\sum_j \sqrt{q_j}$	$(\sum_i \sqrt{q_i})^2/\rho$	$\frac{\sum_j \sqrt{q_j}/\sqrt{q_i}}{\rho}$	$\sqrt{q_i}\sum_j \sqrt{q_j}$

Table: Comparison of Uniform, Proportional, and Square-root replication.

How are the replication strategies implemented?

Chord

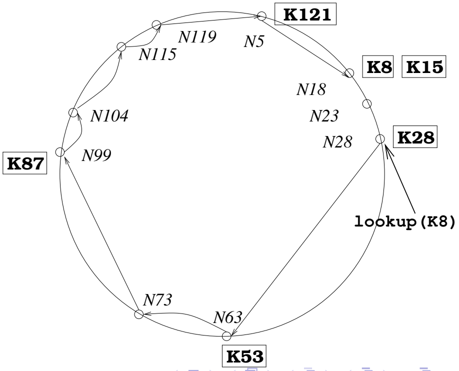
- Node address as well as object value is mapped to a logical identifier in a common flat key space using a consistent hash function.
- When a node joins or leaves the network of n nodes, only $1/n$ keys have to moved.
- Two steps involved.
  - Map the object value to its key
  - Map the key to the node 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 k in the logical space address.



Chord: Simple Lookup

- Each node tracks its successor on the ring.
- A query for key x is forwarded on the ring until it reaches the first node whose identifier $y \geq$ key $x \, mod(2^m)$.
- The result, which includes the IP address of the 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 $key \neq i$:
(1a) if $key \in (i, successor]$ then
(1b)      return($successor$)
(1c) else return $successor.Locate\_Successor(key)$.
```



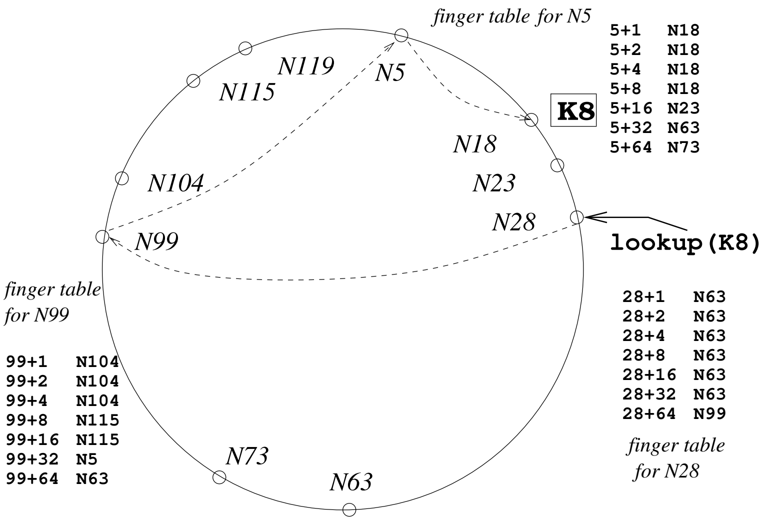
Chord: Scalable Lookup

- Each node i maintains a routing table, called the *finger table*, with $O(log\ n)$ entries, such that the xth entry ($1 \leq x \leq m$) is the node identifier of the node $succ(i + 2^{x-1})$.
- This is 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
- Each finger table entry contains the IP address and port number in addition to the node identifier
- Due to the *log* structure of the finger table, there is more info about nodes closer by than about nodes further away.
- Consider a query on key *key* at node i,
  - if *key* lies between i and its successor, the *key* would reside at the successor and its address is returned.
  - 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.
  - As 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 Code

```
(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]).
```

Chord: Scalable Lookup Example



finger table for N5

5+1	N18
5+2	N18
5+4	N18
5+8	N18
5+16	N23
5+32	N63
5+64	N73

finger table for N99

99+1	N104
99+2	N104
99+4	N104
99+8	N115
99+16	N115
99+32	N5
99+64	N63

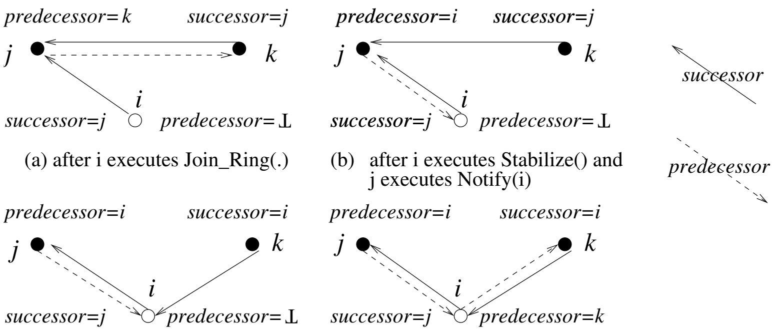
28+1	N63
28+2	N63
28+4	N63
28+8	N63
28+16	N63
28+32	N63
28+64	N99

finger table for N28

Chord: Managing Churn

```
(variables)
integer: $successor \longleftarrow$ initial value;
integer: $predecessor \longleftarrow$ initial value;
array of integer $finger[1 \ldots log\ m]$;
integer: $next\_finger \longleftarrow 1$;


(1) $i.Create\_New\_Ring()$:
(1a) $predecessor \longleftarrow \perp$;
(1b) $successor \longleftarrow i$.

(2) $i.Join\_Ring(j)$, where $j$ is any node on the ring to be joined:
(2a) $predecessor \longleftarrow \perp$;
(2b) $successor \longleftarrow j.Locate\_Successor(i)$.

(3) $i.Stabilize()$:        // executed periodically to verify and inform successor
(3a) $x \longleftarrow successor.predecessor$;
(3b) if $x \in (i, successor)$ then
(3c)     $successor \longleftarrow x$;
(3d) $successor.Notify(i)$.

(4) $i.Notify(j)$:        // $j$ believes it is predecessor of $i$
(4a) if $predecessor = \perp$ or $j \in (predecessor, i)$ then
(4b)     transfer keys in the range $(predecessor, j]$ to $j$;
(4c)     $predecessor \longleftarrow j$.

(5) $i.Fix\_Fingers()$:        // executed periodically to update the finger table
(5a) $next\_finger \longleftarrow next\_finger + 1$;
(5b) if $next\_finger > m$ then
(5c)     $next\_finger \longleftarrow 1$;
(5d) $finger[next\_finger] \longleftarrow Locate\_Successor(i + 2^{next\_finger - 1})$.

(6) $i.Check\_Predecessor()$:        // executed periodically to verify whether predecessor still exists
(6a) if predecessor has failed then
(6b)     $predecessor \longleftarrow \perp$.
```

Chord: Integrating a Node in the Ring



(a) after i executes Join_Ring(.)

(b) after i executes Stabilize() and j executes Notify(i)

(c) after k executes Stabilize(), that triggers step (d)

(d) after i executes Notify(k)

Node i integrates into the ring, where $j > i > k$, as per the steps shown.

Chord

- How are node departures handled? or node failures?
- 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 $O(log\, n)$.
- The search for a successor in *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 $log(n) \leq m$.
- The average lookup time is $1/2\, log(n)$.
- Details of Complexity derivations: refer text.

Content Addressable Network (CAN)

- An indexing mechanism that maps objects to locations in CAN
- object-location in P2P networks, large-scale storage management, wide-area name resolution services that decouple name resolution and the naming scheme
- Efficient, scalable addition of and location of objects using location-independent names or keys.
- 3 basic operations: insertion, search, deletion of $(key, value)$ pairs
- d-dimensional logical Cartesian space organized as a d-torus logical topology, i.e.. d-dimensional mesh with wraparound.
- Space partitioned dynamically among nodes, i.e., node i has space $r(i)$.
- For object v, its key $r(v)$ is mapped to a point $\vec{p}$ in the space. $(v, key(v))$ tuple stored at node which is the present owner containing the point $\vec{p}$. Analogously to retrieve object v.
- 3 components of CAN
  - Set up CAN virtual coordinate space, partition among nodes
  - Routing in virtual coordinate space to locate the node that is assigned the region corresponding to $\vec{p}$
  - Maintain the CAN in spite of node departures and failures.

CAN Initialization

1. Each CAN has a unique DNS name that maps to the IP address of a few bootstrap nodes. Bootstrap node: tracks a partial list of the nodes that it believes are currently in the CAN.
2. A joiner node queries a bootstrap node via a DNS lookup. Bootstrap node replies with the IP addresses of some randomly chosen nodes that it believes are in the CAN.
3. The joiner chooses a random point $\vec{p}$ in the coordinate space. The joiner sends a request to one of the nodes in the CAN, of which it learnt in Step 2, asking to be assigned a region containing $\vec{p}$. The recipient of the request routes the request to the owner $old\_owner(\vec{p})$ of the region containing $\vec{p}$, using CAN routing algorithm.
4. The $old\_owner(\vec{p})$ node splits its region in half and assigns one half to the joiner. The region splitting is done using an a priori ordering of all the dimensions. This also helps to methodically merge regions, if necessary. The (k, v) tuples for which the key k now maps to the zone to be transferred to the joiner, are also transferred to the joiner.
5. The joiner learns the IP addresses of its neighbours from $old\_owner(\vec{p})$. The neighbors are $old\_owner(\vec{p})$ and a subset of the neighbours of $old\_owner(\vec{p})$. $old\_owner(\vec{p})$ also updates its set of neighbours. The new joiner as well as $old\_owner(\vec{p})$ inform their neighbours of the changes to the space allocation, In fact, each node has to send an immediate update of its assigned region, followed by periodic HEARTBEAT refresh messages, to all its neighbours.

When a node joins a CAN, only the neighbouring nodes in the coordinate space are required to participate. The overhead is thus of the order of the number of neighbours, which is $O(d)$ and independent of n.

CAN Routing

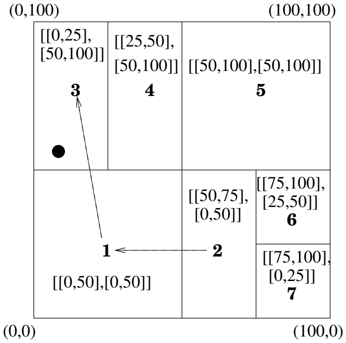
- Straight-line path routing in Euclidean space
- Each node's routing table maintains list of neighbor nodes and their IP addresses virtual Euclidean coordinate regions.
- To locate value v, its key $k(v)$ is mapped to a point $\vec{p}$. Knowing the neighbours' region coordinates, each node follows simple greedy routing by forwarding the message to that neighbour having coordinates that are closest to the destination.

$$argmin_{k \in Neighbours}[\min |\vec{x} - \vec{k}|]$$

- Avg # neighbours of a node is $O(d)$
- Average path length $d/4 \cdot n^{1/d}$.

Advantages over Chord:

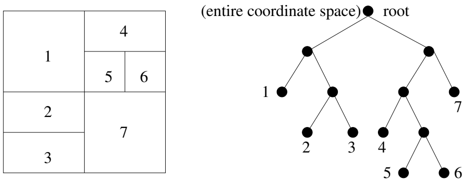
- Each node has about same # neighbors and same amt. of state info, independent of $n \Longrightarrow$ scalability
- Multiple paths in CAN provide fault-tolerance
- Avg path length $n^{1/d}$, not *log n*



Two-dimensional CAN space. Seven regions are shown. The dashed arrows show the routing from node 2 to the coordinate $\vec{p}$ shown by the shaded circle.

CAN Maintainence

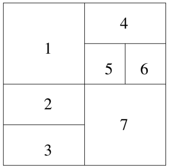
- Voluntary departure: Hand over region and (*key*, *value*) tuples to a neighbor.
- Neighbor choice: formation of a convex region after merger of regions
- Otherwise, neighbor with smallest volume. However, regions are not merged and neighbor handles both regions until background reassignment protocol is run.
- Node failure detected when periodic HEARTBEAT message not received by neighbors. They then run a TAKEOVER protocol to decide which neighbor will own dead node's region. This protocol favors region with smallest volume.
- Despite TAKEOVER protocol, the (*key*, *value*) tuples remain lost until background region reassignment protocol is run.
- Background reassignment protocol: for 1-1 load balancing, restore 1-1 node to region assignment, and prevent fragmentation.

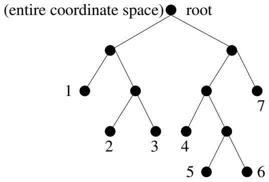


If node 2 fails, its region is assigned to node 3. If node 7 fails, regions 5 and 6 get merged and assigned to node 5 whereas node 6 is assigned the region of the failed node 7.

CAN Maintainence

- Voluntary departure: Hand over region and (*key*, *value*) tuples to a neighbor.
- Neighbor choice: formation of a convex region after merger of regions
- Otherwise, neighbor with smallest volume. However, regions are not merged and neighbor handles both regions until background reassignment protocol is run.
- Node failure detected when periodic HEARTBEAT message not received by neighbors. They then run a TAKEOVER protocol to decide which neighbor will own dead node's region. This protocol favors region with smallest volume.
- Despite TAKEOVER protocol, the (*key*, *value*) tuples remain lost until background region reassignment protocol is run.
- Background reassignment protocol: for 1-1 load balancing, restore 1-1 node to region assignment, and prevent fragmentation.





If node 2 fails, its region is assigned to node 3. If node 7 fails, regions 5 and 6 get merged and assigned to node 5 whereas node 6 is assigned the region of the failed node 7.

CAN Optimizations

Improve per-hop latency, path length, fault tolerance, availability, and load balancing. These techniques typically demonstrate a trade-off.

- **Multiple dimensions.** As the path length is $O(d \cdot n^{1/d})$, increasing the number of dimensions decreases the path length and increases routing fault tolerance at the expense of larger state space per node.
- **Multiple realities or coordinate spaces.** The same node will store different (k, v) tuples belonging to the region assigned to it in each reality, and will also have a different neighbour set. The data contents (k, v) get replicated, leading to higher availability. Furthermore, the multiple copies of each (k, v) tuple offer a choice. Routing fault tolerance also improves.
- Use delay metric instead of Cartesian metric for routing
- Overloading coordinate regions by having multiple nodes assigned to each region. Path length and latency can reduce, fault tolerance improves, per-hop latency decreases.
- Use multiple hash functions. Equivalent to using multiple realities.
- Topologically sensitive overlay. This can greatly reduce per-hop latency.

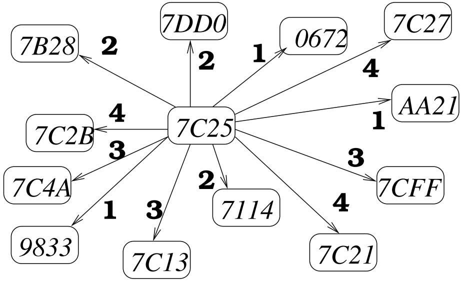
CAN Complexity: $O(d)$ for a joiner. $O(d/4 \cdot log(n))$ for routing. Node departure $O(d^2)$.

Tapestry

- Nodes and objects are assigned IDs from common space via a distributed hashing.
- Hashed node ids are termed VIDs or v_{id}. Hashed object identifiers are termed GUIDs or O_G.
- ID space typically has $m = 160$ bits, and is expressed in hexadecimal.
- If a node v exists such that $v_{id} = O_G$ exists, then that v become the root. If such a v does not exist, then another unique node sharing the largest common prefix with O_G is chosen to be the *surrogate root*.
- The object O_G is stored at the root, or the root has a direct pointer to the object.
- To access object O, reach the root (real or surrogate) using prefix routing
- Prefix routing to select the next hop is done by increasing the prefix match of the next hop's VID with the destination O_{G_R}. Thus, a message destined for $O_{G_R} = 62C35$ could be routed along nodes with VIDs 6****, then 62***, then 62C**, then 62C3*, and then to 62C35.

Tapestry - Routing Table

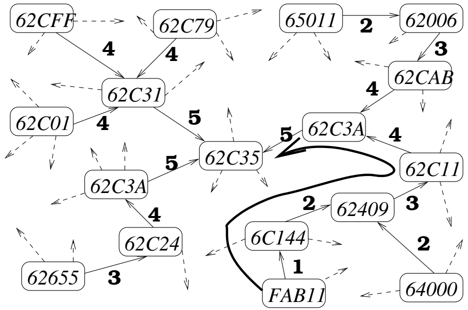
- Let $M = 2^m$. The routing table at node v_{id} contains $b \cdot log_b\, M$ entries, organized in $log_b M$ levels $i = 1 \ldots log_b M$. Each entry is of the form $\langle w_{id}, IP\ address \rangle$.
- Each entry denotes some "neighbour" node VIDs with a $(i-1)$-digit prefix match with v_{id} – thus, the entry's w_{id} matches v_{id} in the $(i-1)$-digit prefix. Further, in level i, for each digit j in the chosen base (e.g., $0, 1, \ldots E, F$ when $b = 16$), there is an entry for which the i^{th} digit position is j.
- For each forward pointer, there is a backward pointer.



Some example links at node with identifier "7C25". Three links each of levels 1 through 4 are labeled.

Tapestry: Routing

- The j^{th} entry in level i may not exist because no node meets the criterion. This is a *hole* in the routing table.
- *Surrogate routing* can be used to route around holes. If the j^{th} entry in level i should be chosen but is missing, route to the next non-empty entry in level i, using wraparound if needed. All the levels from 1 to $log_b 2^m$ need to be considered in routing, thus requiring $log_b 2^m$ hops.



An example of routing from FAB11 to 62C35. The numbers on the arrows show the level of the routing table used. The dashed arrows show some unused links.

Tapestry: Routing Algorithm

- Surrogate routing leads to a unique root.
- For each v_{id}, the routing algorithm identifies a unique spanning tree rooted at v_{id}.

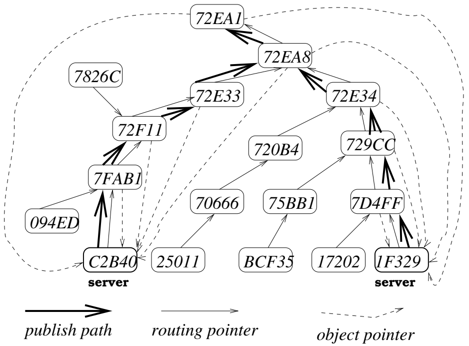
```
(variables)
array of array of integer $Table[1 \ldots log_b 2^m, 1 \ldots b]$;          // routing table

(1) $NEXT\_HOP(i, O_G = d_1 \circ d_2 \ldots \circ d_{log_b M})$ executed at node $v_{id}$ to route to $O_G$:
    // $i$ is (1 + length of longest common prefix), also level of the table
(1a) while $Table[i, d_i] = \perp$ do   // $d_j$ is $i$th digit of destination
(1b)      $d_i \longleftarrow (d_i + 1)$ mod $b$;
(1c) if $Table[i, d_i] = v$ then    // node $v$ also acts as next hop (special case)
(1d)      return $NEXT\_HOP(i + 1, O_G)$   // locally examine next digit of destination
(1e) else return($Table[i, d_i]$).    // node $Table[i, d_i]$ is next hop
```

The logic for determining the next hop at a node with node identifier v, $1 \leq v \leq n$, based on the i^{th} digit of O_G.

Tapestry: Object Publication and Object Search

- The unique spanning tree used to route to v_{id} is used to publish and locate an object whose unique root identifier O_{G_R} is v_{id}.
- A server S that stores object O having GUID O_G and root O_{G_R} periodically publishes the object by routing a *publish* message from S towards O_{G_R}.
- At each hop and including the root node O_{G_R}, the *publish* message creates a pointer to the object
- This is the directory info and is maintained in *soft-state*.
- To search for an object O with GUID O_G, a client sends a query destined for the root O_{G_R}.
    - Along the $log_b 2^m$ hops, if a node finds a pointer to the object residing on server S, the node redirects the query directly to S.
    - Otherwise, it forwards the query towards the root O_{G_R} which is guaranteed to have the pointer for the location mapping.
- A query gets redirected directly to the object as soon as the query path overlaps the publish path towards the same root.

Tapestry: Object Publication and Search



An example showing publishing of object with identifier 72EA1 at two replicas 1F329 and C2B40. A query for the object from 094ED will find the object pointer at 7FAB1. A query from 7826C will find the object pointer at 72F11. A query from BCF35 will find the object pointer at 729CC.

Tapestry: Node Insertions

- For any node Y on the path between a publisher of object O and the root G_{O_R}, node Y should have a pointer to O.
- Nodes which have a hole in their routing table should be notified if the insertion of node X can fill that hole.
- If X becomes the new root of existing objects, references to those objects should now lead to X.
- The routing table for node X must be constructed.
- The nodes near X should include X in their routing tables to perform more efficient routing.

Refer to book for details of the insertion algorithm that maintains the above properties.

Tapestry: Node Deletions and Failures

Node deletion

- Node *A* informs the nodes to which it has (routing) backpointers. It also provides them with replacement entries for each level from its routing table. This is to prevent holes in their routing tables. (The notified neighbours can periodically run the nearest neighbour algorithm to fine-tune their tables.)
- The servers to which *A* has object pointers are also notified. The notified servers send object republish messages.
- During the above steps, node *A* routes messages to objects rooted at itself to their new roots. On completion of the above steps, node *A* informs the nodes reachable via its backpointers and forward pointers that it is leaving, and then leaves.

Node failures: Repair the object location pointers, routing tables and mesh, using the redundancy in the Tapestry routing network. Refer to the book for the algorithms

Tapestry: Complexity

- A search for an object expected to take $(log_b 2^m)$ hops. However, the routing tables are optimized to identify nearest neighbour hops (as per the space metric). Thus, the latency for each hop is expected to be small, compared to that for CAN and Chord protocols.
- The size of the routing table at each node is $c \cdot b \cdot log_b 2^m$, where c is the constant that limits the size of the neighbour set that is maintained for fault-tolerance.

The larger the Tapestry network, the more efficient is the performance. Hence, better if different applications share the same overlay.

Fairness in P2P systems

Selfish behavior, free-riding, leaching degrades P2P performance. Need incentives and punishments to control selfish behavior.

Prisoners' Dilemma

Two suspects, A and B, are arrested by the police. There is not enough evidence for a conviction. The police separate the two prisoners, and separately, offer each the same deal: if the prisoner testifies against (betrays) the other prisoner and the other prisoner remains silent, the betrayer gets freed and the silent accomplice get a 10 year sentence. If both testify against the other (betray), they each receive a 2 year sentence. If both remain silent, the police can only sentence both to a small 6-month term on a minor offense.

Rational selfish behavior: both betray the other - is not Pareto optimal (does not ensure max good for all). Both staying silent is Pareto-optimal but that is not rational. In the iterative version, memory of past moves can be used, in this case, Pareto-optimal solution is reachable.

Tit-for-tat in BitTorrent

Tit-for-tat strategy: first step, you cooperate; in subsequent steps, reciprocate the action done by the other in the previous step.

- The BitTorrent P2P system has adopted the tit-for-tat strategy in deciding whether to allow a download of a file in solving the leaching problem.
- cooperation is analogous to allowing others to upload local files,
- betrayal is analogous to not allowing others to upload.
- *chocking* refers to the refusal to allow uploads.

As the interactions in a P2P system are long-lived, as opposed to a one-time decision to cooperate or not, *optimistic unchocking* is periodically done to unchoke peers that have been chocked. This optimistic action roughly corresponds to the re-initiation of the game with the previously chocked peer after some time epoch has elapsed.

Trust/ reputation in P2P Systems

- Incentive-based economic mechanisms to ensure maximum cooperation inherently depend on the notion of trust.
- P2P environment where the peer population is highly transient, there is also a need to have trust in the quality of data being downloaded.
- This requirements have lead to the area of trust and trust management in P2P systems.
- As no node has a complete view in the P2P system, it may have to contact other nodes to evaluate the trust in particular offerers.
- These communication protocol messages for trust management may be susceptible to various forms of malicious attack (such as man-in-the-middle attacks and Sybil attacks), thereby requiring strong security guarantees.
- The many challenges to tracking trust in a distributed setting include:
  - quantifying trust and using different metrics for trust,
  - how to maintain trust about other peers in the face of collusion,
  - how to minimize the cost of the trust management protocols.

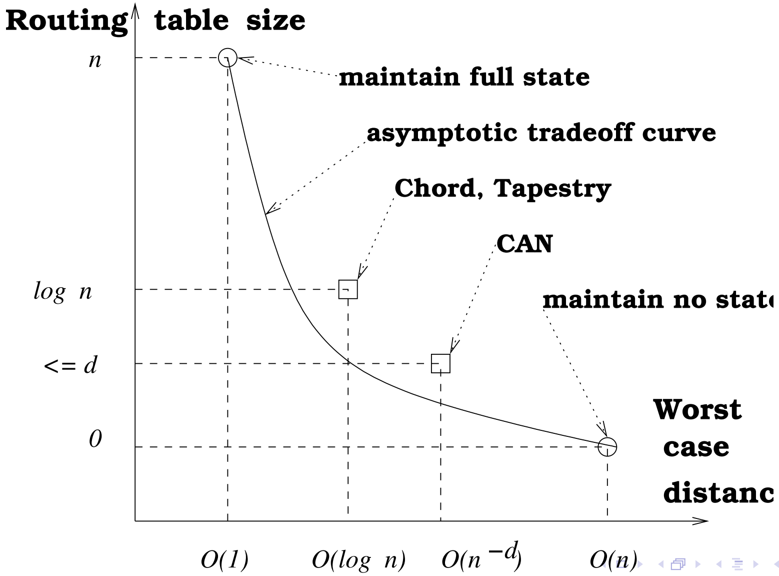
Unifying P2P Protocols

Let the k entries in a routing table at a node with identifier id be the tuples $\langle S_{id,i}, J_{id,i} \rangle$, for $1 \leq i \leq k$. If $|dest - id| \in$ the range $S_{id,i}$ then route to $R(id + J_{id,i})$, where $R(x)$ is the node responsible for key $R(x)$.

Protocol	Chord	CAN	Tapestry
Routing table size	$k = O(log_2 n)$	$k = O(d)$	$k = O(log_b n)$
Worst case distance	$O(log_2 n)$	$O(n^{1/d})$	$O((b-1) \cdot log_b n)$
n, common name space	2^k	x^d	b^x
S_i	$[2^{i-1}, 2^i)$	$[x^{i-1}, x^i)$	$[j \cdot b^{x-lvl+1}, (j+1) \cdot b^{x-lvl+1})$
J_i	2^{i-1}	kx^{i-1}	$suffix(J_{(lvl-1) \cdot b+j}, x - lvl + 1)$

Table: Comparison of representative P2P overlays. d is the number of dimensions in CAN. b is the base in Tapestry.

Asymptotic Tradeoffs between Router Table Size and Network Diameter



Characterizing Complex Networks

Characterize how large networks grow in a distributed (uncontrolled) manner

- WWW, INTNET, AS, SOC, PHON, ACT, AUTH, CITE, WORDOCC, WORDSYN, POWER, PROT, SUBSTRATE

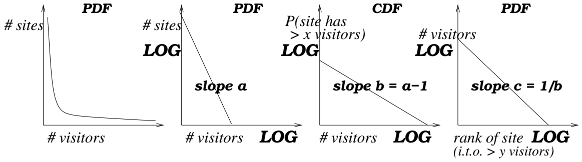
Identify organizational principles that are encoded in subtle ways.

- The Erdos-Renyi (ER) random graph model - n nodes with a link between each pair with probability p is too naive. Three other interesting ideas:
- `Small world model:` Even in very large networks, the path length between any pair of nodes is relatively small. This principle was popularized by sociologist Stanley Milgram by the "six degrees of separation" uncovered between any two people.
- Social networks have cliques, characterized by `clustering coefficients`. For example, consider node i having k_i out-edges. Let l_i be the actual number of edges among the k_i neighbors. Then i's clustering coefficient is $C_i = 2l_i/(k_i(k_i - 1))$. The network clustering coefficient is the average of such C_is. (The clustering coefficient of the ER model is p)
- Let $P(k)$ be the probability that a randomly selected node has k incident edges. In many networks, $P(k) \sim k^{-\gamma}$, i.e., $P(k)$ is distributed with a power-law tail. Such networks that are free of any characteristic scale, i.e., whose degree characterization is independent of n, are called `scale-free networks`. In a random graph, the degree distribution is Poisson-distributed Thus, random graphs are not scale-free.

Some Basic Laws

Consider popularity of web sites as example.

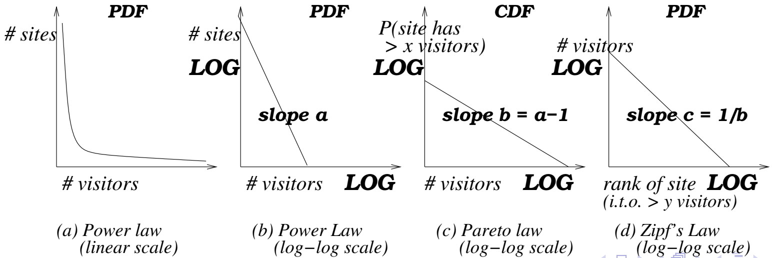
- *PowerLaw* : $P[X = x] \sim x^{-a}$. the number of occurrences of events that equal x is an inverse power of x. In the web example, this corresponds to the number of sites which have exactly x visitors. In the log-log plot, the slope is a.
- Pareto Law: $P[X \geq x] \sim x^{-b} = x^{-(a-1)}$. The number of occurrences larger than x is an inverse power of x. In the web example, the graph corresponds to the number of sites which have at least x visitors. The CDF can be obtained by integrating the PDF. The exponents a and b of the Pareto (CDF) and Power Laws (PDF) are related as $b + 1 = a$. In the log-log plot, the slope is $b = a - 1$.
- Zipf's Law: $n \sim r^{-c}$. This law states the count n (i.e., the number) of the occurrences of an event, as a function of the event's rank r. It says that the count of the rth largest occurrence is an inverse power of the rank r. In the web example, this corresponds to the number of visits to the rth most popular site. In the log-log plot, the slope is c, which can be seen to be $\frac{1}{b} = \frac{1}{a-1}$.



(a) Power law (linear scale) (b) Power Law (log−log scale) (c) Pareto law (log−log scale) (d) Zipf's Law (log−log scale)

Review of Basic Laws (contd.)

- The Pareto Law (CDF) and Power law (PDF) are related.
- Zipf Law $n \sim r^{-c}$, stating "the r-ranked object has $n = r^c$ occurrences, can be equivalently expressed as: "r objects (X-axis) have $n = r^{-c}$ (Y-axis) of more occurrences". This becomes Pareto Law's CDF after transposing the X and Y axes, i.e., by restating as: "the number of occurrences larger than $n = r^{-c}$ (Y-axis) is r (X-axis)".
- From Zipf's Law, $n = r^{-c}$, hence, $r = n^{-\frac{1}{c}}$. Hence, the Pareto exponent b is $\frac{1}{c}$. As $b = (a - 1)$, where a is the Power Law exponent, we see that $a = 1 + \frac{1}{c}$. Hence, the Zipf distribution also satisfies a Power Law PDF.



Properties of the Internet

Rank exponent/ Zipf's law: The nodes in the Internet graph are ranked in decreasing order of their degree. When the degree d_i is plotted as a fn of the rank r_i on a log-log scale, the graph is like Figure (d). The slope is termed the rank exponent $\mathcal{R}$, and $d_i \propto r_i^{\mathcal{R}}$. If the minimum degree $d_n = m$ is known, then $m = d_n = Cn^{\mathcal{R}}$, implying that the proportionality constant C is $m/n^{\mathcal{R}}$.

Degree exponent/ PDF and CDF: Let the CDF f_d of the node degree d be the fraction of nodes with degree greater than d. Then $f_d \propto d^{\mathcal{D}}$, where $\mathcal{D}$ is the degree exponent that is the slope of the log-log plot of f_d as a fn of d.

Analogously, let the PDF be g_d. Then $g_d \propto d^{\mathcal{D}'}$, where $\mathcal{D}'$ is the degree exponent that is the slope of the log-log plot of g_d as a function of d.

Empirically, $D' \sim D + 1$, as theoretically predicted. Further, $\mathcal{R} \sim \frac{1}{\mathcal{D}}$, also as theoretically predicted.

Eigen exponent $\mathcal{E}$: For the adjacency matrix A of a graph, its eigenvalue λ is the solution to $AX = \lambda X$, where X is a vector of real numbers. The eigenvalues are related to the graph's number of edges, number of connected components, the number of spanning trees, the diameter, and other topological properties. Let the various eigenvalues be λ_i, where i is the order and between 1 and n. Then the graph of λ_i as a fn of i is a straight line, with a slope of $\mathcal{E}$, the eigen-exponent. Thus, $\lambda_i \propto i^{\mathcal{E}}$. When the eigenvalues and the degree are sorted in descending order, it is found that $\lambda_i = \sqrt{d_i}$, implying that $\mathcal{E} = \frac{\mathcal{D}}{2}$.

Further properties of the Internet are given in the book.

Classification of Scale-free Networks

Many scale-free networks have degree between 2 and 3.

"betweenness centrality" metric

For any graph, let its geodesics, i.e., set of shortest paths, between any pair of nodes i and j be denoted $S(i,j)$. Let $S_k(i,j)$ be a subset of $S(i,j)$ such that all the geodesics in $S_k(i,j)$ pass through node k. The betweenness centrality BC of node k, b_k, is $\sum_{i \neq j} g_k(i,j) = \sum_{i \neq j} \frac{|S_k(i,j)|}{|S(i,j)|}$. The b_k denotes the importance of node k in shortest-path connections between all pairs of nodes in the network.

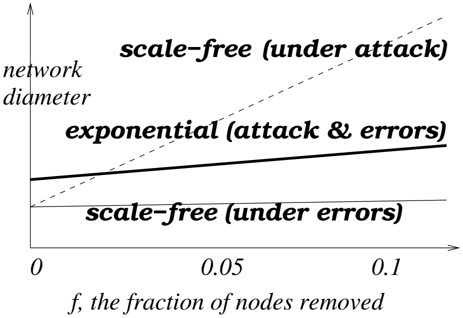
The metric BC follows the power law $P_{BC}(g) \sim g^{-\beta}$, where β is the BC-exponent. Unlike the degree exponent which varies across different network types, the BC-exponent has been empirically found to take on values of only 2 or 2.2 for these network types.

Error and Attack Tolerance

Two types of small-world networks: exponential (EX) and scale-free (SF)

- In EX models, (such as ER and Waltz-Strogatz), $P(k)$ reaches a max then exponentially decreases. In SF, $P(k)$ decreases as per Power Law $P(k) \sim k^{-\gamma}$.
- In EX, nodes with a high degree are almost impossible. In SF, nodes with high degree are statistically significant.
- In EX, all nodes have about the same number of links. In SF, some nodes have many links while the majority have few links.

Errors simulated by removing nodes randomly. Attacks simulated by removing nodes with highest degrees. The relative impact of errors and attacks on the `network diameter` is shown.



Impact on Network Partitioning: EX Networks

The impact of removal of nodes on partitioning is measured using two metrics: S_{max}, the ratio of the size of the largest cluster to the system size, and S_{others}, the average size of all clusters except the largest.

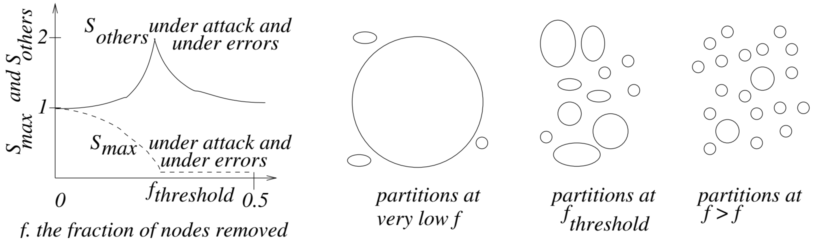
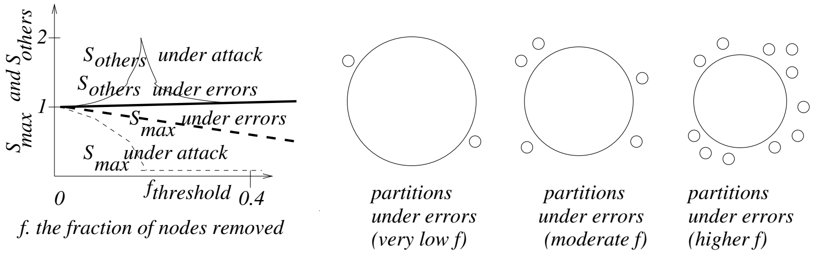


Fig 18.13. Impact on cluster size of exponential networks, from Albert, Jeong, and Barabasi. (a) Graphical trend. (b) Pictorial cluster sizes for low f, i.e., $f \ll f_{threshold}$. (c) Pictorial cluster sizes for $f \sim f_{threshold}$. (d) Pictorial cluster sizes for $f > f_{threshold}$. The pictorial trend in (b)-(d) is also exhibited by scale-free networks under attack, but for a lower value of $f_{threshold}$.

Impact of Network Partitioning: SF Networks



Impact on cluster size of scale-free networks, from Albert, Jeong, and Barabasi. The pictorial impact of attacks on cluster sizes are similar to those in Fig 18.13. (a) Graphical trend. (b) Pictorial cluster sizes for low f under failure. (c) Pictorial cluster sizes for moderate f under failure. (d) Pictorial cluster sizes for high f under failure.

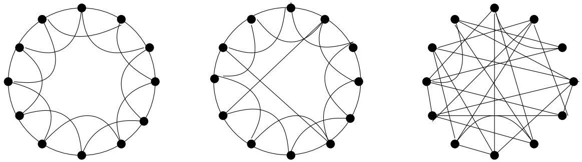
Generalized Random Graph Networks

- Random graphs are not scale-free (node deg distribution does not follow Power Law).
- The generalized random graph uses the degree distribution as an input (i.e., requires the degree distribution to follow a power law), but is random in all other respects.
- These semi-random graphs have a random distribution of edges, similar to ER model.
- Clustering coefficient tends to 0 as n increases.

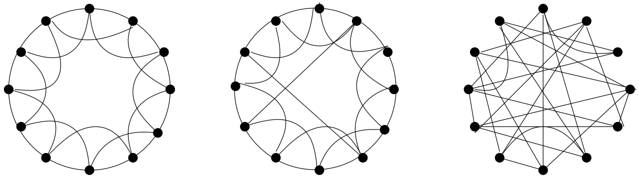
Small World Networks

Real-world networks are small worlds, having small diameter, like random graphs, but they have relatively large clustering coefficients that tend to be independent of the network size.

Ordered lattices have clustering coefficients independent of the network size. Figure(a) shows a 1-D lattice in which each node is connected to $k = 4$ closest nodes. The clustering coefficient $C = \frac{3(k-2)}{4(k-1)}$.


(a) *(b)* *(c)*

Small Worlds Networks: Watts-Strogatz model



(a) *(b)* *(c)*

The Watts-Strogatz random rewiring procedure. (a) Regular. (b) Small-world. (c) Random. The rewiring shown maintains the degree of each node. The Watts-Strogatz model is the first model for small world graphs with high clustering coefficients with low path lengths.

1. Define a ring lattice with n nodes and each node connected to k closest neighbours ($k/2$ on either side). Let $n \gg k \gg ln(n) \gg 1$.
2. Rewire each edge randomly with probability p. When $p = 0$, there is a perfect structure, as in Figure(b). When $p = 1$, complete randomness, as in Figure(c).

Small World Networks (2)

A characteristic of small-world graphs is the small average path length. When p is small, len scales linearly with n but when p is large, len scales logarithmically. Through analytical arguments and simulations, the characteristic path length tends to vary as:

$$len(n, p) \sim \frac{n^{1/d}}{k} f(pkn) \tag{5}$$

where the function f behaves as follows.

$$f(u) = \begin{cases} constant & \text{if } u \ll 1 \\ \frac{ln(u)}{u} & \text{if } u \gg 1 \end{cases} \tag{6}$$

The variable $u \propto pkn^d$ has the intuitive interpretation that it depends on the average number of random links that provide "jumps" across the graph, and $f(u)$ is the average factor by which the distance between a pair of nodes gets reduced by the "jumps".

Scale-free Networks

Semi-random graphs that are constrained to obey a power law for the degree distributions and constrained to have large clustering coefficients yield scale-free networks, but do not shed any insight into the mechanisms that give birth to scale-free networks.

Initially, there are m_0 isolated nodes. At each sequential step, perform one of the following operations.

Growth: Add a new node with m edges ($m \leq m_0$), that link the new node to m nodes already in the system.

Preferential Attachment: The probability $\prod$ that the new node will be connected to node i depends on the degree k_i such that $\prod(k_i) = \frac{k_i}{\sum_j (k_j)}$

The simple Barabasi-Albert model has a degree distribution having a power law (degree 3) that is independent of m.

Evolving Networks

For a flexible, universal model, need to incorporate

- Preferential attachment: Model the fact that a new node attaches to an isolated node.
- Growth: # edges increases faster than the # nodes
- Local events: Model microscopic (local) changes to topology, like node addition or deletion, edge addition and deletion
- Growth constraints: Model bounded capacity for #edges or lifetime of nodes. Thus model bounded capacity and aging.
- Competition: A node may attract or inherit more edges at cost of other nodes
- Induced preferential attachment: Local mechanisms, such as copying (web sites) or tracing selected walks (in citation networks) induce preferential attachment.

Extended Barabasi-Albert Model

Initially, there are m_0 isolated nodes. At each sequential step, perform one of the following operations.

With probability p, add m, where $m \leq m_0$, new edges. For each new edge, one end is randomly selected, the other end with probability

$$\prod(k_i) = \frac{k_i + 1}{\sum_j (k_j + 1)} \tag{7}$$

With probability q, rewire m edges. To rewire an edge, randomly select node i, delete some edge (i, w), add edge (i, x) to node x that is chosen with probability $\prod(k_x)$ as per Equation 7.

With probability $1 - p - q$, insert a new node. Add m new edges to the new node, such that with probability $\prod(k_i)$, an edge connects to a node i already present before this step.

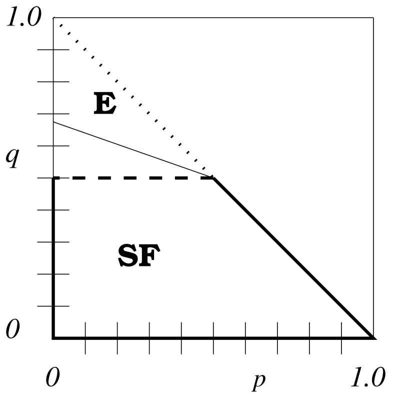
Extended Barabasi-Albert Model

$q < q_{max}$: The degree distribution is a power law.

$q > q_{max}$: $P(k)$ can be shown to behave like an exponential distribution. The model now behaves like the ER and WS models.

This is similar to the behaviour seen in real networks – some networks show a power law while others show an exponential tail – and a single model can capture both behaviors by tuning the parameter q.

The scale-free regime and the exponential regime are marked. The boundary between the two regimes depends on the value of m and has slope $-m/(1+2m)$. The area enclosed by thick lines shows the scale-free regime; the dashed line is its boundary when $m \to \infty$ and the dotted line is its boundary when $m \to 0$.



Phase diagram for the Extended Barabasi-Albert model. **SF** denotes the scale-free regime, which is enclosed by the thick border. **E** denotes the exponential regime which exists in the remainder of the lower diagonal region of the graph. The plain line shows the boundary for $m = 1$, having a Y-axis intercept at 0.67.