CHAPTER

18 Peer-to-peer computing and overlay graphs

18.1 Introduction

Peer-to-peer (P2P) network systems use an application-level organization of the network overlay for flexibly sharing resources (e.g., files and multimedia documents) stored across network-wide computers. In contrast to the client–server model, any node in a P2P network can act as a server to others and, at the same time, act as a client. Communication and exchange of information is performed directly between the participating peers and the relationships between the nodes in the network are equal. Thus, P2P networks differ from other Internet applications in that they tend to share data from a large number of end users rather than from the more central machines and Web servers. Several well known P2P networks that allow P2P file-sharing include Napster [25], Gnutella [16,17], Freenet [10], Pastry [30], Chord [32], and CAN [27].

Traditional distributed systems used DNS (domain name service) to provide a lookup from host names (logical names) to IP addresses. Special DNS servers are required, and manual configuration of the routing information is necessary to allow requesting client nodes to navigate the DNS hierarchy. Further, DNS is confined to locating hosts or services (not data objects that have to be a priori associated with specific computers), and host names need to be structured as per administrative boundary regulations. P2P networks overcome these drawbacks, and, more importantly, allow the location of arbitrary data objects.

An important characteristic of P2P networks is their ability to provide a large combined storage, CPU power, and other resources while imposing a low cost for scalability, and for entry into and exit from the network. The ongoing entry and exit of various nodes, as well as dynamic insertion and deletion of objects is termed as *churn*. The impact of churn should be as transparent as possible. P2P networks exhibit a high level of self-organization and are able to operate efficiently despite the lack of any prior infrastructure or authority. The philosophy of this model requires that if a node wants to

Table 18.1 Desirable characteristics and performance features of P2P systems.

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

enjoy the services which other nodes provide, that node should provide service to other nodes. Some desirable features of P2P systems are summarized in Table 18.1.

18.1.1 Napster

One of the earliest popular P2P systems, Napster [25], used a server-mediated central index architecture organized around clusters of servers that store direct indices of the files in the system. The 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. The basic steps of operation to search for content and to determine a node from which to download the content are the following:

1. A client connects to a meta-server that assigns a lightly loaded server from one of the close-by clusters of servers to process the client's query.
2. The client connects to the assigned server and forwards its query along with its own identity.
3. The server responds to the client with information about the users connected to it and the files they are sharing.
4. 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.

18.1.2 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-pont (representing a logical all-to-all connectivity) once a connection is established.

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

18.2 Data indexing and overlays

The data in a P2P network is identified by using indexing. Data indexing allows the physical data independence from the applications. Indexing mechanisms can be classified as being *centralized*, *local*, or *distributed*:

- **Centralized indexing** entails the use of one or a few central servers to store references (indexes) to the data on many peers. The DNS lookup as well as the lookup by some early P2P networks such as Napster used a central directory lookup.
- **Distributed indexing** involves the indexes to the objects at various peers being scattered across other peers throughout the P2P network. In order to access the indexes, a structure is used in the P2P overlay to access the indexes. Distributed indexing is the most challenging of the indexing schemes, and many novel mechanisms have been proposed, most notably the *distributed hash table (DHT)*. Various DHT schemes differ in the hash mapping, search algorithms, diameter for lookup, search diameter, fault-tolerance, and resilience to churn.

  A typical DHT uses a flat key space to associate the mapping between network nodes and data objects/files/values. Specifically, the node address is mapped to a logical identifier in the key space using a consistent hash function. The data object/file/value is also mapped to the same key space using hashing. These mappings are illustrated in Figure 18.1.
- **Local indexing** requires each peer to index only the local data objects and remote objects need to be searched for. This form of indexing is typically used in unstructured overlays in conjunction with flooding search or random walk search. Gnutella uses local indexing.

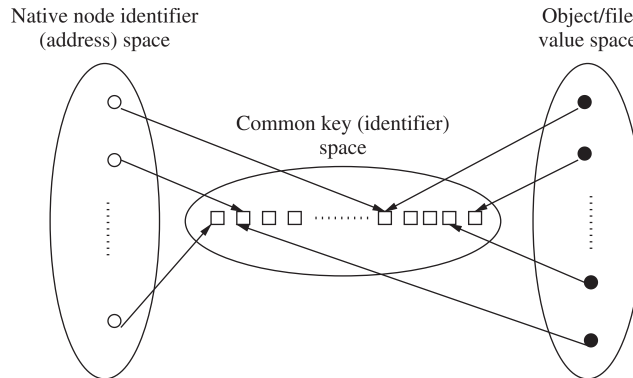

Figure 18.1 The mappings from node address space and object space in a typical DHT scheme, e.g., Chord, CAN, Tapestry.

An alternate way to classify indexing mechanisms is as being a *semantic index mechanism* or a *semantic-free* index mechanism. A semantic index is human readable, for example, a document name, a keyword, or a database key. A semantic-free index is not human readable and typically corresponds to the index obtained by a hash mechanism, e.g., the DHT schemes. A semantic index mechanism supports keyword searches, range searches, and approximate searches, whereas these searches are not supported by semantic-free index mechanisms.

18.2.1 Distributed indexing

Structured overlays

The P2P network topology has a definite structure, and the placement of files or data in this network is highly deterministic as per some algorithmic mapping. (The placement of files can sometimes be "loose," as in some earlier P2P systems like Freenet, where "hints" are used.) The objective of such a deterministic mapping is to allow a very fast and deterministic lookup to satisfy queries for the data. These systems are termed as *lookup systems* and typically use a hash table interface for the mapping. The hash function, which efficiently maps *keys* to *values*, in conjunction with the regular structure of the overlay, allows fast search for the location of the file.

An implicit characteristic of such a deterministic mapping of a file to a location is that the mapping can be based on a single characteristic of the file (such as its name, its length, or more generally some *predetermined* function computed on the file). A disadvantage of such a mapping is that arbitrary queries, such as range queries, attribute queries and exact keyword queries cannot be handled directly.

Another implicit effect of the tight coupling of the regular overlay structure and the rigid mapping function to enable fast access is that file insertions and deletions incur some overhead which may be nontrivial under churn.

Unstructured overlays

The P2P network topology does not have any particular controlled structure, nor is there any control over where files/data is placed. Each peer typically indexes only its local data objects, hence, *local indexing* is used. Node joins and departures are easy – the local overlay is simply adjusted. File placement is not governed by the topology. Search for a file may entail high message overhead and high delays. However, complex queries are supported because the search criteria can be arbitrary.

Although the P2P network topology does not have any controlled structure, some topologies naturally emerge. The following topologies are common and will be studied in later sections:

- **Power law random graph** (PLRG) This is a random graph where the node degrees follow the power law. Here, if the nodes are ranked in terms of their degree, then the ith node has c/i^{α} neighbors, where c is a constant.
- **Normal random graph** This is a normal random graph where the nodes typically have a uniform degree.

We study search in unstructured overlay networks in the next section.

18.3 Unstructured overlays

18.3.1 Unstructured overlays: properties

Unstructured overlays have the serious disadvantage that queries may take a long time to find a file or may be unsuccessful even if the queried object exists. The message overhead of a query search may also be high.

The following are the main advantages of unstructured overlays such as the one used by Gnutella:

- Exact keyword queries, range queries, attribute-based queries, and other complex queries can be supported because the search query can capture the semantics of the data being sought; and the indexing of the files and data is not bound to any non-semantic structure.
- Unstructured overlays can accommodate high churn, i.e., the rapid joining and departure of many nodes without affecting performance.

The following are advantages of unstructured overlays if certain conditions are satisfied:

- Unstructured overlays are efficient when there is some degree of data replication in the network.
- Users are satisfied with a best-effort search.
- The network is not so large as to lead to scalability problems during the search process.

18.3.2 Gnutella

Gnutella uses a fully decentralized architecture [16, 17]. In Gnutella logical overlays, nodes index only their local content. The acutal overlay topology can be arbitrary as nodes join and leave randomly. A node joins the Gnutella network by forming a connection to some nodes found in standard Gnutella directory-like databases. (Note that the function of joining the network cannot be said to be fully decentralized.) Users communicate with each other, performing the role of both *serv*er and cli*ent*, termed as *servent*. The following are the main message types used by Gnutella:

- *Ping* messages are used to discover hosts, and allow a new host to announce itself.
- *Pong* messages are the responses to *Ping*s. The *Pong* messages indicate the port and (IP) address of the responder, and some information about the amount of data (the number and size of files) that node can make available.
- *Query* messages. The search strategy used is flooding. *Query* messages contain a search string and the minimum download speed required of the potential responder, and are flooded in the network.
- *QueryHit* messages are sent as responses if a node receiving a *Query* detects a local match in response to a query. A *QueryHit* contains the port and address (IP), speed, the number of files found, and related information. The path traced by a *Query* is recorded in the message, so the *QueryHit* follows the same path in reverse.

18.3.3 Search in Gnutella and 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 [23],

$$\sum_{i=1}^{m} q_i = 1, \tag{18.1}$$

$$\text{uniform: } q_i = 1/m; \qquad \text{Zipf-like: } q_i \propto i^{-\alpha}. \tag{18.2}$$

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,

$$\sum_{i=1}^{m} r_i = R; \qquad p_i = r_i/R, \tag{18.3}$$

$$\text{uniform: } r_i = R/m; \quad \text{proportional: } r_i \propto q_i; \quad \text{square-root: } r_i \propto \sqrt{q_i}. \tag{18.4}$$

Under uniform replication, all objects have an equal number of replicas and 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.

For an object search, some of the more popular metrics of efficiency are:

- the 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, and hence, each search is inherently independent. In guided search, nodes store some history of past searches to aid future searches. Various mechanisms for caching hints to guide and narrow down future searches are used. In this chapter, we focus on unguided searches in the context of unstructured overlays.

Search strategies

Flooding [23]

- In order to curtail the high message overhead that flooding introduces, the initial strategy was to use *checking*. Here, a node checks back with the query originator before forwarding a query. Unfortunately, this cause heavy load on the originator, in addition to excessive delays, and hence is not practical.
- The next approach is to use the *time to live* (TTL) field or the hop count. However, this does not guarantee that a match can be found for the query even if the object exists in the network, and requires a high value of TTL to have a high degree of success.
- A refinement that allows more control is the *expanding ring* strategy. A node first floods with a small TTL. If the search is not successful, it starts another flood with a larger TTL, and so on. This strategy is more successful when objects are replicated.

  The expanding ring approach is significantly more successful than the TTL approach, for all replication strategies, and all query distributions, and the cost is only a relatively small increase in delay.

Although the expanding ring is superior to TTL, both are flooding-based strategies and suffer from message duplication.

Random walk

Another strategy to use is that of *random walking*. Here, a query is randomly forwarded by a node when it is received. Random walk greatly reduces the message overhead but it increases the search latency. Hence, *k random walkers* can be used. To terminate the *k* random walkers, a "checking-cum-TTL" strategy is effective. Here, each walker periodically (after a certain number of hops) checks with the query originator whether to terminate; the TTL is used to prevent looping, and is usually set to a large value.

Performance

The performance of searches in unstructured overlays has been studied via simulations and by experiments. The following are some of the relationships of interest, for both flooding and for *k*-random walk (for various values of *k*) for various graph topologies such as the random graph and the PLRG:

- 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 as a function of various replication ratios and replication strategies.

Guidelines

- Adaptively determining the termination condition is important. Checking is adaptive whereas TTL is not.
- Message duplication must be minimized, as it represents wasted resources.
- At each step in the search, the number of messages (or number of nodes visited) should not increase by a large amount.

Overall, *k*-random walk performs much better than flooding and is more scalable, for various replication and query distributions, and various graph topologies.

18.3.4 Replication strategies

Cohen and Shenker [12] studied the degree of replication for blind or *unguided* search in random overlay graphs. The various parameters used to study replication are defined in Table 18.2. Random search is modeled by the following process. A node is repeatedly drawn at random from a bin, examined for a match with the copy of the object, and replaced in the bin, until the object is found. The metric then is the number of nodes drawn (or equivalently, the

Table 18.2 Parameters to study replication.

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

number of hops of a random walker) until success. The probability that the object is found on the kth drawing is:

$$Pr_i(k) = \frac{r_i}{n}(1 - \frac{r_i}{n})^{k-1}.$$

The average search size for i, denoted as A_i, is:

$$A_i = E_{over\ all\ k}(Pr_i(k)) = \sum_{k=1}^{n}\left[k\frac{r_i}{n}\left(1 - \frac{r_i}{n}\right)^{k-1}\right] \sim \frac{n}{r_i},\ \text{for large } n. \quad (18.5)$$

Across the system, the average search size A is:

$$\text{average search size } A = \sum_{i=1}^{m} q_i A_i = n\sum_i \frac{q_i}{r_i}. \quad (18.6)$$

Setting r_i to n maximizes A, but requires full replication. As resources are constrained, assume that average number of replicas per node is $\rho = R/n < m$. (It is easy to see that $R \geq m \geq \rho$.) Substituting for n with R/ρ in the equation above, we have:

$$\text{average search size } A = \frac{R}{\rho}\sum_i \frac{q_i}{r_i} = \frac{1}{\rho}\sum_i \frac{q_i}{p_i}. \quad (18.7)$$

The *utilization rate* u_i of a replica of object i is the average rate of requests serviced by a replica of i. With random search, $u_i = q_i/p_i = R(q_i/r_i)$. Over all replicas of object i, the utilization is simply $= Rq_i$. The average utilization rate over (all copies of) all objects is $u = \sum_{i=1}^{m} r_i(u_i/R) = \sum_{i=1}^{m} p_i(q_i/p_i) = 1$. This average is a constant, and independent of the replication scheme. It is desirable to have a low maximum utilization rate in order to distribute the load more uniformly.

The replication problem is formulated as the optimization solution for Eq. (18.7). We assume that all objects are of uniform size. To simplify analysis, we also assume that each object that is queried exists in the system and a search continues until the object is found, i.e., all searches are eventually successful. (In practice, there is a parameter L – such as TTL – that controls

the maximum search size. Search on insoluble queries continues until this parameter is exceeded. The cost of such queries is $f_s A + (1 - f_s)L$, where f_s is the fraction of queries that are soluble.)

Two natural replication strategies are *uniform* and *proportional*:

- **Uniform** $r_i = R/m$, which implies $p_i = r_i/R = 1/m$.
  The average search size for object i is $A_i = n/r_i$. This equals $R/(\rho r_i) = R/(\rho R/m_i) = m/\rho$ and is the same for all objects.
  From Eq. (18.7), the average search size $A_{uniform} = (1/\rho)\sum_i (q_i/p_i) = \frac{1}{\rho}\sum_i mq_i = (m/\rho)$.
  The utilization of a replica of i is $u_i = q_i/p_i$, which is proportional to the query rate as p_i is the same for all objects.
  The maximum utilization of a replica of i is $max_i u_i = max_i(q_i/p_i) = R(q_i/r_i)$, which can vary significantly.
- **Proportional** $r_i = Rq_i$, which implies $p_i = q_i$.
  The average search size for object i is $A_i = n/r_i = n/Rp_i = n/(Rq_i) = 1/(\rho q_i)$, which is inversely proportional to the query rate.
  From Eq. (18.7), the average search size $A_{proportional} = (1/\rho)\sum_i (q_i/p_i) = (1/\rho)\sum_{i=1}^{m} 1 = m/\rho$.
  The utilization of a replica of i is $u_i = q_i/p_i = 1$, a constant for all replicas of all objects.
  The maximum utilization of a replica of i is $max_i u_i = max_i(q_i/p_i) = max_i(q_i/q_i) = 1$ for all i.

Both uniform and proportional replication have the same average search size, which is independent of the query distribution. However, objects whose query rates are below the average have lower overhead with uniform replication, while those with query rates larger than the average have lower overhead with proportional replication.

- **Square root** The optimal replication strategy that minimizes the average search size is the square-root replication, which is defined as having $p_i = r_i/R \propto \sqrt{q_i}/\sum_j \sqrt{q_j}$, assuming that $1/R \le \sqrt{q_i}/\sum_j \sqrt{q_j} \le n/R$ for all i.
  The optimality of square-root replication can be seen as follows. Substituting $1 - \sum_{i=1}^{m-1} p_i$ for p_m in the cost function of Eq. (18.7), we have:

$$\text{search size } A_{sq-rt} = \frac{1}{\rho}\sum_i q_i/p_i = \frac{1}{\rho}\left[\sum_{i=1}^{m-1} q_i/p_i + q_m \bigg/ \left(1 - \sum_{i=1}^{m-1} p_i\right)\right].$$

  By solving $ds/dp_i = 0$, the value of p_i that minimizes A_{sq-rt} is seen to be $p_m\sqrt{q_i/q_m}$.

Analogous to uniform and proportional replications, the values of A, A_i, and u_i for square-root replication can be dervied. Exercise 18.1 asks you to show the derivations. The results are summarized in Table 18.3. It can be seen that to minimize A, $r_i = R\sqrt{q_i}/\sum_j \sqrt{q_j}$.

Table 18.3 Comparison of uniform, proportional, and square-root replication [23].

	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	$R\sqrt{q_i}/\sum_j \sqrt{q_j}$	$(\sum_i \sqrt{q_i})^2/\rho$	$\frac{\sum_j \sqrt{q_j}/\sqrt{q_i}}{\rho}$	$\sqrt{q_i}\sum_j \sqrt{q_j}$

The square-root replication rate ($\propto \sqrt{q_i}$) is more than that of uniform ($\propto 1$), but less than that of proportional ($\propto q_i$). It has been shown that:

- any allocation rate "in between" that of uniform and of proportional has a lower average search size A than that of uniform and proportional;
- any allocation rate either less than that of uniform, or greater than that of proportional has a higher average search size A than that of uniform and proportional.

18.3.5 Implementing replication strategies

Proportional and uniform can be trivally implemented. For proportional, each query creates a copy; for uniform, a fixed number of copies are made when an object is created [12].

The simple "path replication" scheme, wherein the number of copies made is proportional to the length of the (successful) search path, implements square-root replication. Here object i is replicated $c(n/r_i)$ times per query, where c is some constant. Then r_i can be captured by the following equation: $dr_i/dt = q_i c(n/r_i)$. Let $a = ln(r_i/r_j)$, then:

$$\frac{da}{dt} = cn\left(\frac{q_j}{r_j^2} - \frac{q_i}{r_i^2}\right) = \frac{1}{r_j}\frac{dr_j}{dt} - \frac{1}{r_i}\frac{dr_i}{dt}.$$

Square-root replication, wherein $r_i = (R/\sum\sqrt{q_i})\sqrt{q_i}$, is a fixed-point solution of this equation. Therefore, path replication implements square-root replication.

The analysis implicitly assumes that replicas also get deleted, in a way that is independent of their object identity or query rate, and the lifetime of a replica is a non-decreasing function of its age. (Policies such as random and FIFO satisfy this condition, but LRU and LFU do not.) Then, during steady state, the creation rate can equal the deletion rate.

An alternate way of analyzing replication schemes is as follows. Let C be the number of replicas created on a successful query; $\overline{C}$ is its average. Then, in steady state,

$$\frac{p_i}{p_j} = \frac{q_i\overline{C_i}}{q_j\overline{C_j}}. \tag{18.8}$$

To implement distributed algorithms for various replication policies, it is necessary to determine C_i locally without knowing p_i or q_i:

- For proportional replication, C is the same for all objects.
- For square-root replication, if $\overline{C_i} \propto 1/\sqrt{q_i}$ then $p_i/p_j = \sqrt{q_i/q_j}$, by substituting in Eq. (18.8).
  As $A_i \propto nR/p_i$ and $p_i \propto q_i\overline{C_i}$, therefore $A_i \propto 1/(q_i\overline{C_i})$.
  With *path replication*, $C_i \propto A_i$, hence $C_i \propto A_i \propto 1/(q_i\overline{C_i})$.
  In steady state, A_i and $\overline{C_i}$ are equal. Solving $C_i \propto 1/(q_i\overline{C_i})$ for the fixed point, $C_i \propto 1/\sqrt{q_i}$. As $p_i \propto q_iC_i$ when C_i is steady, this gives $p_i \propto \sqrt{q_i}$. In a practical implementation, it needs to be ensured that convergence occurs once steady state sets in.

18.4 Chord distributed hash table

18.4.1 Overview

The Chord protocol, proposed by Stoica *et al.* [32], uses a flat key space to associate the mapping between network nodes and data objects/files/values. The node address as well as the data object/file/value is mapped to a logical identifier in the common key space using a consistent hash function. These mappings are illustrated in Figure 18.1. Both these mappings should ensure that the keys are distributed roughly equally among the nodes. This also insures that with high probability, the overhead of key management when nodes join or leave the P2P network is low. Specifically, when a node joins or leaves the network having n nodes, only $O(1/n)$ keys need to be moved from one location to another.

The Chord key space is flat, thus giving applications flexibility in mapping their files/data to keys. Chord supports a single operation, $lookup(x)$, which maps a given key x to a network node. Specifically, Chord stores a file/object/value at the node to which the file/object/value's key maps. Two steps are involved:

1. Map the object/file/value to its key in the common address space.
2. Map the key to the node in its native address space using $lookup$. The design of $lookup$ is the main challenge.

In Chord, a node's IP address is hashed to an m-bit identifier that serves as the node identifier in the common key (identifier) space. Similarly, the file/data key is hashed to an m-bit identifier that serves as the key identifier. m is sufficiently large so that the probability of collisions during the hash is negligible. The Chord overlay uses a logical ring of size 2^m. The identifier space is ordered on the logical ring modulo 2^m. Henceforth in this section, we will assume modulo 2^m arithmetic. A key k gets assigned to the first node such that its node identifier equals or follows the key identifier of k in the

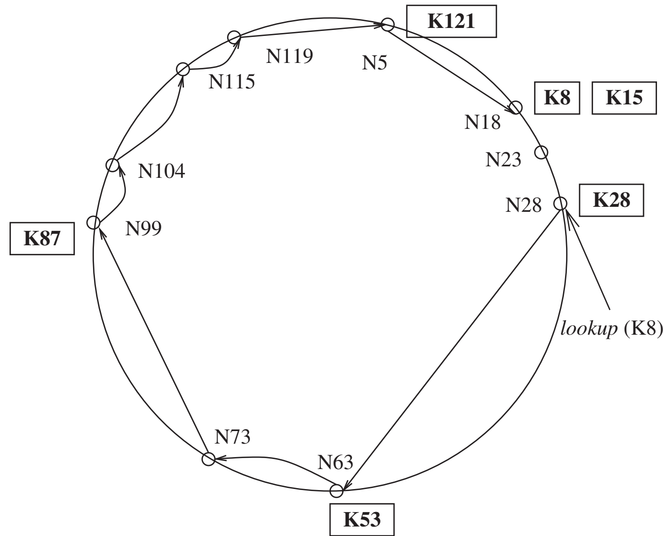

Figure 18.2 An example Chord ring with $m = 7$, showing mappings to the Chord address space, and a query lookup using a simple scheme [32].

common identifier space. The node is the successor of k, denoted $succ(k)$. A Chord ring for $m = 7$ is depicted in Figure 18.2. 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 as follows: $succ(8) = 18$, $succ(15) = 18$, $succ(28) = 28$, $succ(53) = 63$, $succ(87) = 99$, and $succ(121) = 5$.

18.4.2 Simple lookup

A simple key lookup algorithm that requires each node to store only 1 entry in its routing table works as follows. Each node tracks its successor on the ring, in the variable *successor*; a query for key x is forwarded to the successors of nodes until it reaches the first node such that that node's identifier y is greater than the key x, modulo 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, where n is the number of nodes in the P2P network. The pseudo-code for this simple lookup is given in Algorithm 18.1. The following convention is assumed. Notation $(x, y]$ represents the left-open right-closed segment of the Chord logical ring modulo m. Notation $x.Proc(\cdot)$ is a RPC to execute *Proc* on node x while $x.var$ is a RPC to read the variable *var* at process x.

Example The steps for the query: lookup(K8) initiated at node 28, are shown in Figure 18.2 using arrows.

(variables)
integer: *successor* $\longleftarrow$ initial value;

(1) $i.Locate_Successor(key)$, where $key \neq i$:
(1a) **if** $key \in (i, successor]$ **then**
(1b) **return**(*successor*)
(1c) **else return** $(successor.Locate_Successor(key))$.

Algorithm 18.1 A simple object location algorithm in Chord at node *i* [32].

18.4.3 Scalable lookup

A scalable lookup algorithm that uses $O(log\, n)$ message hops at the cost of $O(m)$ space in the local routing tables, uses the following idea. Each node *i* maintains a routing table, called the *finger table*, with at most $O(log\, n)$ distinct entries, such that the *x*th 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 $2^{x-1}\, mod\, 2^m$. Note that each finger table entry would have to contain the IP address and port number in addition to the node identifier, in order that *i* can communicate with $i.finger[x]$; henceforth we will assume this implicitly without showing these entries.

The size of the finger table is bounded by *m* entries. Due to the logarithmic structure, the finger table has more information about nodes closer ahead of it in the Chord overlay, than about nodes further away. Given any key whose node is to be located, the highly scalable logarithmic search shown in Algorithm 18.2 is used. For a query on key *key* at node *i*, if *key* lies between

(variables)
integer: *successor* $\longleftarrow$ initial value;
integer: *predecessor* $\longleftarrow$ initial value;
integer $finger[1...m]$;

(1) $i.Locate_Successor(key)$, where $key \neq i$:
(1a) **if** $key \in (i, successor]$ **then**
(1b) **return**(*successor*)
(1c) **else**
(1d) $j \longleftarrow Closest_Preceding_Node(key)$;
(1e) **return** $(j.Locate_Successor(key))$.

(2) $i.Closest_Preceding_Node(key)$, where $key \neq i$:
(2a) **for** $count = m$ **down to** 1 **do**
(2b) **if** $finger[count] \in (i, key]$ **then**
(2c) **break**();
(2d) **return**$(finger[count])$.

Algorithm 18.2 A scalable object location algorithm in Chord at node *i* [32].

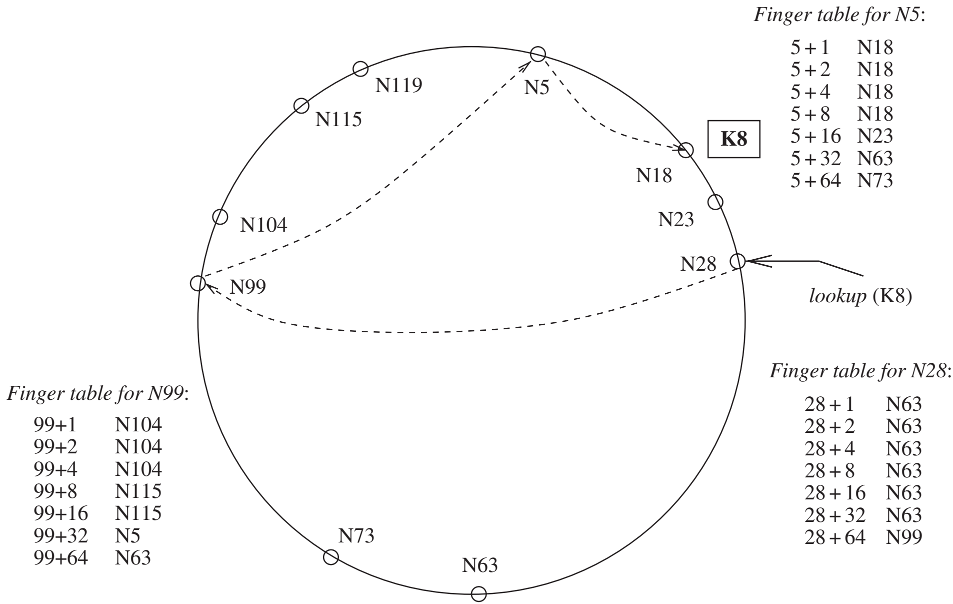

Figure 18.3 An example showing a query lookup using the logarithmically-structured finger tables [32].

i and its successor, the key would reside at the successor and the successor's 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.

Example The use of the finger tables in answering the query lookup(K8) at node N28 is illustrated in Figure 18.3. The finger tables of N28, N99, and N5 that are used are shown.

18.4.4 Managing Churn

The code to manage dynamic node joins, departures, and failures is given in Algorithm 18.3.

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

```
(variables)
integer: successor ⟵ initial value;
integer: predecessor ⟵ initial value;
integer finger[1...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).

(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 ⟵ ⊥.
```

Algorithm 18.3 Managing churn in Chord. Code shown is for node *i* [32].

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 tables 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.

Figure 18.4 illustrates the main steps of the joining process. A recent joiner node i that has executed *Join_Ring*$(\cdot)$ gets integrated into the ring by the following sequence:

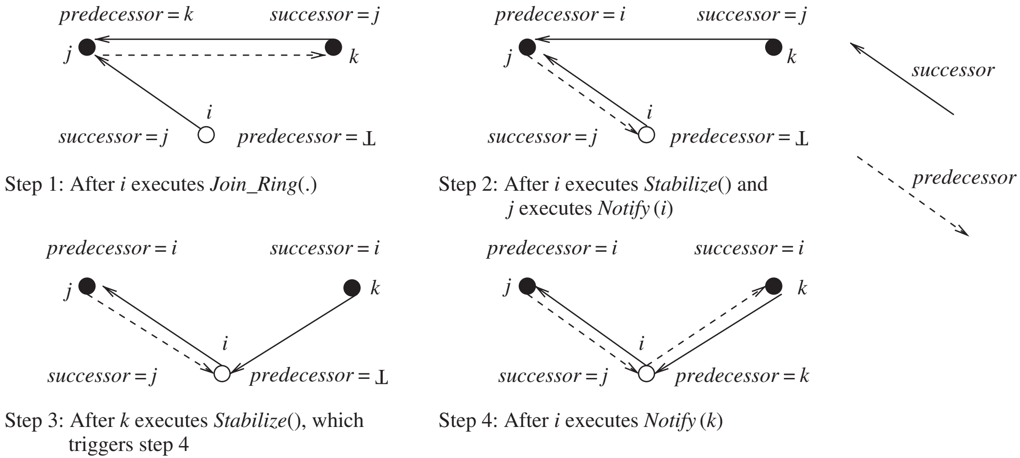

Figure 18.4 Steps in the integration of node i in the ring, where $j > i > k$ [32].

1. The configuration after a recent joiner node i has executed $Join_Ring(\cdot)$.
2. Node i executes $Stabilize()$, which allows its successor j to adjust j's variable $predecessor$ to i. Specifically, when node i invokes $Stabilize()$, it identifies the successor's predecessor k. If $k \in (i, successor)$, then i updates its $successor$ to k. In either case, i notifies its successor of itself via $successor.Notify(i)$, so the successor has a chance to adjust its $predecessor$ variable to i.
3. The earlier predecessor k of j (i.e., the predecessor in Step 1) executes $Stabilize()$ and adjusts its $successor$ pointer from j to i.
4. Node i executes $Fix_Fingers()$ to build its finger table, and other nodes also execute the procedure to update their finger tables if necessary.

Once all the successor variables and finger tables have stabilized, a call by any node to $Locate_Successor(\cdot)$ will reflect the new joiner i. Until then, a call to $Locate_Successor(\cdot)$ may result in the $Locate_Successor(\cdot)$ 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.

Showing the correctness of the Chord protocol in the face of concurrent join operations and stablize operations in which pointers are being rewired is non-trivial. It can be shown that for any set of concurrent join operations, at some point after the last join operation completes, all the pointers and finger tables will be correct. However, in the transient period before the Chord ring stabilizes, an object search can result in three outcomes:

- The finger tables used in a search are up to date and the correct successor of the key is sought in $O(log\, n)$ hops.

- The finger tables are not up to date but the successor pointers are correct. The sought key will be located but may take more steps as the full advantage of a logarithmic search space pruning cannot be used.
- If the successor pointers are incorrect, or the key transfer to the new joiners in procedure *Notify* has not completed, the search may fail. This is during a transient duration, and the source has the choice of reissuing the query.

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 18.2 that knowing that the successor is functional, and that the nodes pointed to by the finger pointers are functional, is essential.

Example In Figure 18.3, assume that node N63 fails. The closest successor that node N28 can find via the finger table is N99. N73 cannot be detected, and keys K64 through K73 will effectively be lost.

A solution such as introducing a *Check_Successor*() procedure analogous to *Check_Predecessor* procedure will not solve the problem because it does not help to identify the functional successor. The Chord protocol proposes that, rather than maintain a single successor, each node maintains a list of α successors, which are the node's first α successors. If the first successor does not respond, the node can try the next successor from the list, and so on. Only the simultaneous failure of all the α successors can then cause the protocol to fail. Maintaining a list of successors requires some changes to the code in Algorithm 18.3. Exercise 18.2 asks you to adapt this code to the changes required for maintaining successor lists.

The provision for a successor list at each node provides a natural mechanism for the application to manage replicated objects. The replicas get placed at the node corresponding to the object key, as well as at the nodes in the successor list of that node. As Chord is able to update its successor list as the successor list changes, Chord can also interface with the application to let it track the locations of the replicas.

A voluntary departure from the ring can be treated as a failure. However, a failed node causes all the data (keys) stored at that node to be lost until corrective action is taken. When a node departs voluntarily, it should first transfer all the keys it is responsible for to its successor. The departing node should also inform its successor and predecessor. This will enable the successor to update its predecessor to the predecessor of the departing node.

The predecessor will also be able to update its successor list by deleting the departing node and adding the last successor of the departing node's successor list to its own successor list.

18.4.5 Complexity

The following results on the complexity have a non-trivial correctness proof and interested readers should consult the Chord papers for the proofs.

1. 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 "high probability" clause is required because the validity of the result depends on the randomness and conflict-free mappings of the hash function used.
2. The search for a successor in *Locate_Successor* in a Chord network with n nodes requires time complexity $O(log\, n)$ with high probability.

   This result is based on the observation that assuming completely random distributions of the key mappings and node mappings, after $2\, log\, n$ hops, the distance between the key being searched for and the present node that the query has reached is at most $1/n$.
3. The size of the finger table is $log(n) \leq m$.
4. The average lookup time is $1/2\, log(n)$.

Exercises 18.2 and 18.3, based on the Chord papers, ask you to prove further results about the complexity under churn conditions.

18.5 Content addressible networks (CAN)

18.5.1 Overview

A content-addressible network (CAN) is essentially an indexing mechanism that maps objects to their locations in the network. The CAN project originated from the observation that the bottleneck to designing a scalable P2P network is this indexing mechanism. An efficient and scalable CAN is useful not only for object location in P2P networks, but also for large-scale storage management systems and wide-area name resolution services that decouple name resolution and the naming scheme. All these applications inherently require efficient and scalable addition of and location of objects using arbitrary location-independent names or keys for the objects.

A CAN supports three basic operations: insertion, search, and deletion of (*key, value*) tuples. (A "value" is an object in the context of a CAN.) A good CAN design is distributed, fault-tolerant, scalable, independent of the

naming structure, implementable at the application layer, and *autonomic*, i.e., self-organizing and self-healing. Although CAN is a generic phrase, it also specifically denotes the particular design of a CAN proposed by Ratnasamy *et al.* [27]. We now study this particular CAN design.

CAN is a *logical d-dimensional Cartesian coordinate space organized as a d-torus logical topology*, i.e., a virtual overlay d-dimensional mesh with wrap-around. A two-dimensional torus was shown in Figure 1.5(a) in Chapter 1. The entire space is partitioned *dynamically* among all the nodes present, so that each node i is assigned a disjoint region $r(i)$ of the space. As nodes arrive, depart, or fail, the set of participating nodes, as well as the assignment of regions to nodes, change.

For any object v, its key $k(v)$ is mapped using a deterministic hash function to a point $\vec{p}$ in the Cartesian coordinate space. The (k, v) pair is stored at the node that is presently assigned the region that contains the point $\vec{p}$. In other words, the (k, v) pair is stored at node i if presently the point $\vec{p}$ corresponding to (k, v) lies in region $r(i)$. Analogously, to retrieve object v, the same hash function is used to map its key k to the same point $\vec{p}$. The node that is presently assigned the region that contains $\vec{p}$ is accessed (using a CAN routing algorithm) to retrieve v. The three core components of a CAN design are the following:

1. Setting up the CAN virtual coordinate space, and partitioning it among the nodes as they join the CAN.
2. Routing in the virtual coordinate space to locate the node that is assigned the region containing $\vec{p}$.
3. Maintaining the CAN due to node departures and failures.

18.5.2 CAN initialization

1. Each CAN is assumed to have a unique DNS name that maps to the IP address of one or a few bootstrap nodes of that CAN. A bootstrap node is responsible for tracking a partial list of the nodes that it believes are currently participating in the CAN. These are reasonable assumptions, and perhaps the most "non-distributed" portions of the CAN design.
2. To join a CAN, the joiner node queries a bootstrap node via a DNS lookup, and the bootstrap node replies with the IP addresses of some randomly chosen nodes that it believes are participating 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 the 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, so as to decide which dimension to split along. 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 neighbors from $old_owner(\vec{p})$. The neighbors are $old_owner(\vec{p})$ and a subset of the neighbors of $old_owner(\vec{p})$. $old_owner(\vec{p})$ also updates its set of neighbors. The new joiner as well as $old_owner(\vec{p})$ inform their neighbors of the changes to the space allocation, so that they have correct information about their neighborhood and can route correctly. In fact, each node has to send an immediate update of its assigned region, followed by periodic HEARTBEAT refresh messages, to all its neighbors.

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

18.5.3 CAN routing

CAN routing uses the straight-line path from the source to the destination in the logical Euclidean space. This routing is realized as follows. Each node maintains a routing table that tracks its neighbor nodes in the logical coordinate space. In d-dimensional space, nodes x and y are neighbors if the coordinate ranges of their regions overlap in $d-1$ dimensions, and abut in one dimension. All the regions are *convex* and can be characterized as follows. Let $region(x) = [[x^1_{min}, x^1_{max}], \ldots, [x^d_{min}, x^d_{max}]]$. Let $region(y) = [[y^1_{min}, y^1_{max}], \ldots, [y^d_{min}, y^d_{max}]]$. Nodes x and y are neighbors if there is some dimension j such that $x^j_{max} = y^j_{min}$ and for all other dimensions i, $[x^i_{min}, x^i_{max}]$ and $[y^i_{min}, y^i_{max}]$ overlap. An example of neighbouring nodes in two-dimensional space is shown in Figure 18.5.

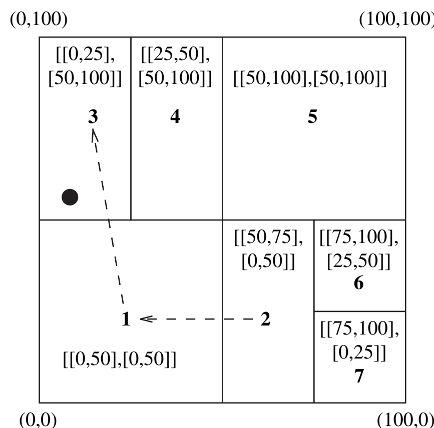

Figure 18.5 Two-dimensional CAN space. Seven regions are shown. The dashed arrows show the routing from node 2 to the coordinate $\vec{p}$ shown by the shaded circle [27].

The routing table at each node tracks the IP address and the virtual coordinate region of each neighbor. To locate value v, its key $k(v)$ is mapped to a point $\vec{p}$ whose coordinates are used in the message header. Knowing the neighbors' region coordinates, each node follows simple greedy routing by forwarding the message to that neighbor having coordinates that are closest to the destination's coordinates. To implement greedy routing to a destination node x, the present node routes a message to that neighbor among the neighbors $k \in Neighbors$, given by

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

Here, $\vec{x}$ and $\vec{k}$ are the coordinates of nodes x and k.

Assuming equal-sized zones in d-dimensional space, the average number of neighbors for a node is $O(d)$. The average path length is $(d/4) \cdot n^{1/d}$. The implication on scaling is that each node has about the same number of neighbors and needs to maintain about the same amount of state information, irrespective of the total number of nodes participating in the CAN. In this respect, the CAN structure is superior to that of Chord. Also note that unlike in Chord, there are typically many paths for any given source-destination pair. This greatly helps for fault-tolerance. Average path length in CAN scales as $O(n^{1/d})$ as opposed to $log\, n$ for Chord.

18.5.4 CAN maintainence

When a node voluntarily departs from CAN, it hands over its region and the associated database of $(key, value)$ tuples to one of its neighbors. The neighbor is chosen as follows. If the node's region can be merged with that of one of its neighbors to form a valid convex region, then such a neighbor is chosen. Otherwise the node's region is handed over to the neighbor whose region has the smallest volume or load – the regions are not merged and the neighbor handles both zones temporarily until a periodic background region reassignment process runs to integrate the regions and prevent further fragmentation.

CAN requires each node to periodically send a HEARTBEAT update message to each neighbor, giving its assigned region coordinates, the list of its neighbors, and their assigned region coordinates. When a node dies, the neighbors suspect its death and initiate a TAKEOVER protocol to decide who will take over the crashed node's region. Despite this TAKEOVER protocol, the $(key, value)$ tuples in the crashed node's database remain lost until the primary sources of those tuples refresh the tuples. Requiring the primary sources to periodically issue such refreshes also serves the dual purpose of updating stale (dirty) objects in the CAN.

The TAKEOVER protocol is as follows. When a node suspects that a neighbor has died, it starts a timer in proportion to its region's volume.

On timeout, it sends a TAKEOVER message, with its region volume piggybacked on the message, to all the neighbors of the suspected failed node. When a TAKEOVER message is received, a node cancels its bid to take over the failed node's region if the received TAKEOVER message contains a smaller region volume than that of the recipient's region. This protocol thus helps in load balancing by choosing the neighbor whose region volume is the smallest, to take over the failed node's region. As all nodes initiate the TAKEOVER protocol, the node taking over also discovers its neighbors and vica versa. In the case of multiple concurrent node failures in one vicinity of the Cartesian space (this is rare), a more complex protocol using a expanding ring search for the TAKEOVER messages can be used.

A graceful departure as well as a failure can result in a neighbor holding more than one region if its region cannot be merged with that of the departed or failed node. To prevent the resulting fragmentation and restore the $1 \rightarrow 1$ node to region assignment, there is a background reassignment algorithm that is run periodically. Conceptually, consider a binary tree whose root represents the entire space. An internal node represents a region that existed earlier but is now split into regions represented by its children nodes. A leaf represents a currently existing region, and (overloading the semantics and the notation), also the node that represents that region.

When a leaf node x fails or departs, there are two cases:

1. If its sibling node y is also a leaf, then the regions of x and y are merged and assigned to y. The region corresponding to the parent of x and y becomes a leaf and it is assigned to node y.
2. If the sibling node y is not a leaf, run a depth-first search in the subtree rooted at y until a pair of sibling leaves (say, $z1$ and $z2$) is found. Merge the regions of $z1$ and $z2$, making their parent z a leaf node, assign the merged region to node $z2$, and the region of x is assigned to node $z1$.

Figure 18.6 illustrates this reassignment. 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.

A distributed version of the above depth-first centralized tree traversal can be performed by the neighbors of a departed node. The distributed traversal leverages the fact that when a region is split, it is done in accordance to a

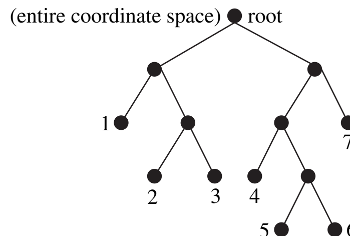

Figure 18.6 Example showing region reassignment in a CAN [27].

particular ordering on the dimensions. Node i performs its part of the depth-first traversal (initiated by the node to which the region of the departed node x is assigned in the TAKEOVER protocol) as follows:

1. Identify the highest ordered dimension dim_a that has the shortest coordinate range $[i_{min}^{dim_a}, i_{max}^{dim_a}]$. Node i's region was last halved along dimension dim_a.
2. Identify neighbor j such that j is assigned the region that was split off from i's region in the last partition along dimension dim_a. Node j's region abuts i's region along dimension dim_a.
3. If j's region volume equals i's region volume, the two nodes are siblings and the regions can be combined. This is the terminating case of the depth-first tree search for siblings. Node j is assigned the combined region, and node i takes over the region of the departed node x. This takeover by node i is done by returning the recursive search request to the originator node, and communicating i's identity on the replies.
4. Otherwise, j's region volume must be smaller than i's region volume. Node i forwards a recursive depth-first search request to j.

18.5.5 CAN optimizations

The following design techniques aim to improve one or more of the performance factors: the per-hop latency, the 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** A coordinate space is termed as a *reality*. The use of multiple independent realities assigns to each node a different region in each different reality. This implies that in each reality, the same node will store different (k, v) tuples belonging to the region assigned to it in that reality, and will also have a different neighbor set. The data contents (k, v) get replicated in each reality, leading to higher data availability. Furthermore, the multiple copies of each (k, v) tuple, one in each reality, offer a choice – the closest copy can be accessed. Routing fault tolerance also improves because each reality offers a set of different paths to the same (k, v) tuple. All these advantages come at the cost of more storage – for state information for the neighbors in each reality, as well as for the (k, v) tuples mapped to the region allocated to a node in each reality.
- **Delay latency** Rather than using just the Cartesian distance as a metric to make routing decisions, the delay latency (measured using the round-trip time (RTT)) on each of the candidate logical links can also be used in making the routing decision.
- **Overloading coordinate regions** Each region can be shared by multiple nodes, up to some upper limit. This offers several advantages. First, the

path length and path latency get reduced because overloading is equivalent to having fewer nodes in the CAN. Second, the fault tolerance improves because a region becomes empty only if all the nodes assigned to it depart or fail concurrently. Third, the per-hop latency decreases because a node can select the closest node from the neighboring region to forward a message towards the destination. The cost of gaining these advantages is that many of the aspects of the basic CAN protocol need to be reengineered to accommodate overloading of coordinate regions (see Exercise 18.5).

- **Multiple hash functions** The use of multiple hash functions maps each key to different points in the coordinate space. This replicates each (k, v) pair for each hash function used. The effect is similar to that of using multiple realities.
- **Topologically sensitive overlay** The CAN overlay described so far has no correlation to the physical proximity or to the IP addresses of domains. Logical neighbors in the overlay may be geographically far apart, and logically distant nodes may be physical neighbors. By constructing an overlay that accounts for physical proximity in determining logical neighbors, the average query latency can be significantly reduced.

18.5.6 CAN complexity

The time overhead for a new joiner is $O(d)$ for updating the new neighbors in the CAN, and $O(d/4 \cdot log(n))$ for routing to the appropriate location in the coordinate space. This is also the overhead in terms of the number of messages. The time overhead and the overhead in terms of the number of messages for a node departure is $O(d^2)$, because the TAKEOVER protocol uses a message exchange between each pair of neighbors of the departed node. Exercise 18.4 asks you to compute the complexity of the distributed region reassignment protocol.

18.6 Tapestry

18.6.1 Overview

The Tapestry P2P overlay network provides efficient scalable location-independent routing to locate objects distributed across the Tapestry nodes [20,21,30,36]. Much of the design is adapted from an earlier design of Plaxton trees [26]. The notable enhancements of Tapestry include dealing with node churn as well as dynamic addition and deletion of objects. As in Chord, nodes as well as objects are assigned identifiers obtained by mapping from their native name spaces to a common large identifier space using a uniformly distributed hash function such as SHA-1. The hashed node identifiers are termed VIDs (the acronym for *virtual* i.d.s) and the hashed object identifiers are termed as GUIDs (acronym for *globally unique* i.d.s). For brevity, a specific

node v's virtual identifier is denoted v_{id} and a specific object O's GUID is denoted O_G.

18.6.2 Overlay and routing

Root and surrogate root

Tapestry uses a common identifier space specified using m bit values. This identifier is typically expressed in hexadecimal notation, i.e., base $b = 16$, and presently Tapestry recommends $m = 160$. Each identifier O_G in this common overlay space is mapped to a set of *unique* nodes that exists in the network, termed as the identifier's root set denoted $\mathcal{O}_{G_R}$. Typically, $|\mathcal{O}_{G_R}|$ is a small constant, and the main purpose of having $|\mathcal{O}_{G_R}| > 1$ is to increase fault-tolerance. In our discussion, we assume $|\mathcal{O}_{G_R}| = 1$, and refer to a root node of O_G as O_{G_R}.

If there exists a node v such that $v_{id} = O_{G_R}$, then v is the root of identifier O_G. If such a node does not exist, then a globally known deterministic rule is used to identify another unique node sharing the largest common prefix with O_G, that acts as the *surrogate* root. To access object O, the goal is to reach the root O_{G_R} (whether real or surrogate). Routing to O_{G_R} is done using distributed routing tables that are constructed using *prefix routing* information. Prefix routing in Tapestry is somewhat analogous to prefix routing within the telephone network, or to address allocation in the Internet using classless interdomain routing (CIDR). Unlike the telephone numbers or CIDR-assigned IP addresses, Tapestry's VIDs are in a virtual space without correlation to topology, however, topological information can be used to select nodes that are "close" as per some metric.

Prefix routing

Prefix routing at any node 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. 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 $\langle w_{id}, IP\,address \rangle$. In level i, there are b entries with the following property:

- Each entry denotes some "neighbor" node VIDs with an $(i-1)$-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 ith digit position is j. Specifically, the jth entry (counting from 0) in level i has value j for digit position i. Let an i digit prefix of v_{id} be denoted as $prefix(v_{id}, i)$. Then the jth entry (counting from 0) in level i begins with an i-digit prefix $prefix(v_{id}, i-1) \circ j$. For example, the fifth entry in level 2 at node 9F248 will be 94***, thus having a two-digit prefix "94."

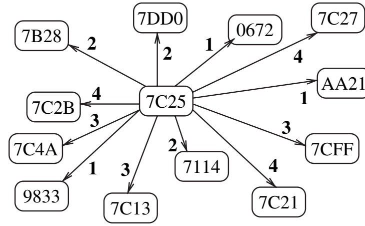

Figure 18.7 Some example links of the Tapestry routing mesh at node with identifier "7C25"[35]. Three links from each level 1 through 4 are labeled by the level.

Router Table

The nodes in the router table at v_{id} are the *neighbors* in the overlay, and these are exactly the nodes with which v_{id} communicates. A part of the routing mesh at one node is shown in Figure 18.7. For each *forward pointer* from node v to v', there is a *backward pointer* from v' to v. Observe the following regarding the router table construction:

- There is a choice of which entry to add in the router table. For example, the jth entry in level i can be the VID of any node whose i-digit prefix is determined; the $(m-i)$-digit suffix can vary. The flexibility is useful to select a node that is "close", as defined by some metric space (e.g., round-trip time). In fact, this choice also allows a more fault-tolerant strategy for routing. Multiple VIDs can be stored in the routing table, as follows. For each prefix β of a node v's identifier and for each digit $j \in \{0, \ldots, b-1\}$ in the alphabet, define the *neighbor set* $\mathcal{N}^v_{\beta,j}$ as the set of all nodes whose identifiers share prefix $\beta \circ j$. The nodes in this neighbor set are also referred to as (β, j) neighbors of v. The b sets, one for each value of j, form the routing table of level $|\beta|+1$. $|\mathcal{N}^v_{\beta,j}|$ grows exponentially as $|\beta|$ decreases, so the size of this set can be limited by a predetermined parameter c. The closest node in each set is the primary neighbor. Thus the size of the routing table is: $c \cdot b \cdot log_b M$.

  The route from v^0_{id} (source) to destination $j_1 \circ j_2 \cdots \circ j_{log\,M}$, is via nodes $v^1, v^2, \ldots, v^{log\,M}$, where $v^1 \in \mathcal{N}^{v^0}_{\perp,j_1}$ (first hop), $v^2 \in \mathcal{N}^{v^1}_{j_1,j_2}$ (second hop), $v^1 \in \mathcal{N}^{v^2}_{j_1 \circ j_2, j_3}$ (third hop), and so on. The primary neighbor is chosen at each hop. Observe that this provides *location-independent* routing, i.e., irrespective of the source, the same unique root node is reached.
- The jth entry in level i may not exist because no node meets the criterion. This is a *hole* in the routing table. Stated more generally, $|\mathcal{N}^v_{\beta,j}|$ may be 0, signifying a hole for digit j at level $|\beta|+1$.

  Surrogate routing can be used to route around holes. If the jth 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. The code for determining the next hop using $NEXT_HOP(i, O_G)$ is shown in Algorithm 18.4. This is invoked as $NEXT_HOP(1, O_G)$ at the source node. To determine hop i of the route, the node v that executes the function has a prefix at least $i-1$ digits in common with O_G.

(variables)
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 ith digit of destination
(1b) $d_i \longleftarrow (d_i + 1) \bmod b$;
(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

Algorithm 18.4 Routing in Tapestry [35]. The logic for determining the next hop at a node with node identifier v, $1 \le v \le n$, based on the ith digit of O_G, i.e., based on the digit in the ith most significant position in O_G.

Example An example of routing is shown in Figure 18.8.

Property 1 Surrogate routing leads to a unique root. If the routing were to lead to different nodes A and B, let the most significant position in which the digits of A and B differ be i. This implies level i routing caused the routing at some nodes X and Y along different digits. However, the first i digits do not change henceforth, and, assuming synchronized routing tables, the holes would be consistent in the tables at X and Y. Hence both should route to the same ith digit, which is a contradiction. It can now be seen that:

Property 2 For each identifier v_{id}, the routing algorithm identifies a unique spanning tree rooted at v_{id}.

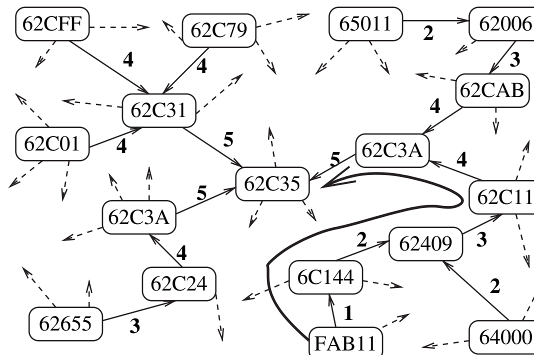

Figure 18.8 An example of routing from FAB11 to 62C35 [35]. The numbers on the arrows show the level of the routing table used. The dashed arrows show some unused links.

18.6.3 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. Ideally, "each node between O and O_{G_R} must maintain a pointer to O despite churn." (Note that the publishing is done by each server at which a replica of the object resides, as well as for each GUID of the object. Recall that an object can be assigned multiple GUIDs, each mapping to a different root node, and giving rise to the set of root nodes $\mathcal{O}_{G_R}$.) If a node lies on the path from two or more servers storing replicas, that node will store a pointer to each replica, sorted in terms of a distance metric (such as latency from itself). This is the directory information for objects, and is maintained as a *soft-state*, i.e., it requires periodic updates from the server, to deal with changes and to provide fault-tolerance.

Example An example showing publishing of an object with O_G = 72EA1 by two replicas, at 1F329 and C2B40 is shown in Figure 18.9.

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. Each hop towards the root reduces the choice of the selection of its next node by a factor of b; hence, the more likely by a factor of b that a query path

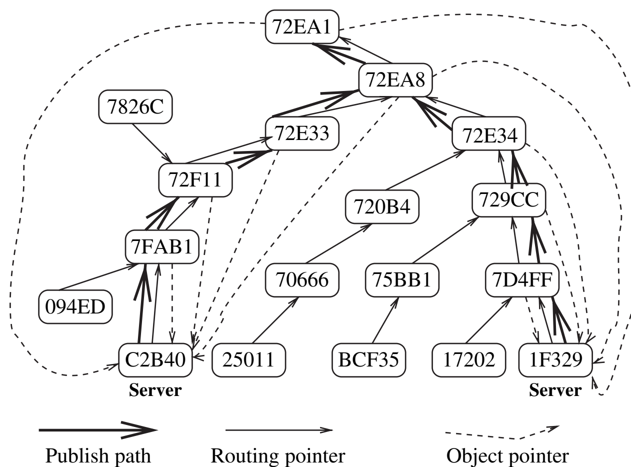

Figure 18.9 An example showing publishing of object with identifier 72EA1 at two replicas 1F329 and C2B40 [35].

and a publish path will meet. Furthermore, as the next hop is chosen based on the network distance metric whenever there is a choice, we also observe that the closer the client is to the server in terms of the distance metric, the more likely that their paths to the object root will meet sooner, and the faster the query will be redirected to the object.

Example Consider the object O_G which has identifier 72EA1 and two replicas at 1F329 and C2B40, as shown in Figure 18.9. 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.

18.6.4 Node insertion

When nodes join the network, the result should be the same as though the network and the routing tables had been initialized with the nodes as part of the network. The procedure for the insertion of node X should maintain the following property of Tapestry:

Property 3 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.

More generally, the insertion should satisfy the following properties:

- Nodes that 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.

The main steps in node insertion are as follows:

1. Node X uses some gateway node into the Tapestry network to route a message to itself. This leads to its "surrogate," i.e., the root node with identifier closest to that of itself (which is X_{id}). The surrogate Z identifies the length α of the longest common prefix that Z_{id} shares with X_{id}.
2. Node Z initiates a MULTICAST-CONVERGECAST on behalf of X by essentially creating a logical spanning tree as follows. Acting as a root, Z contacts all the (α, j) nodes, for all $j \in \{0, 1, \ldots, b-1\}$ (tree level 1). These are the nodes with prefix α followed by digit j. Each such (level 1) node $Z1$ contacts all the $(prefix(Z1, |\alpha|+1), j)$ nodes, for all $j \in \{0, 1, \ldots, b-1\}$ (tree level 2). This continues up to level $log_b 2^m - |\alpha|$ and completes the MULTICAST. The nodes at this level are the leaves

of the tree, and initiate the CONVERGECAST, which also helps to detect the termination of this phase.

All the nodes contacted fill in any holes in their routing table and, if necessary, transfer any references of pointers that are rooted locally. All these nodes also contact X with their information, so that X can build its routing table from level $|\alpha|+1$ up to $log_b 2^m$. All these nodes that contact X have a common prefix of α.

To construct the rest of its routing table from levels 1 through $|\alpha|$, node X procures similar lists for successively smaller prefixes until it gets closest b nodes matching the empty prefix. Node X begins with the list of nodes for level α, corresponding to the level l of its routing table which is already filled. To construct the level $l-1$ list, node X contacts all the nodes in the level l list to find out all the level $l-1$ nodes they know about by asking for both forward pointers and backward pointers. Level $l-1$ of the routing table is filled in using the k closest nodes from the level $l-1$ list, for each of the digits $0, \ldots, b-1$. In this manner, X completes its routing table, and all the nodes contacted in the process can optimize their routing tables by using X if it helps.

The insertion protocols are fairly complex and deal with concurrent insertions.

18.6.5 Node deletion

When a node A leaves the Tapestry overlay, the following actions are performed:

1. 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 neighbors can periodically run the nearest neighbor algorithm to fine-tune their tables.)
2. The servers to which A has object pointers are also notified. The notified servers send object republish messages.
3. 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 are handled by using the redundancy that is built in to the routing tables and object location pointers. For example, each routing table entry has up to c neighbors in the neighbor set $\mathcal{N}^v_{\beta,j}$. A node X detects a failure of another node A by using soft-state beacons or when a node sends a message but does not get a response. Node X updates its routing table entry for A with a suitable substitute node, running the nearest neighbor algorithm if necessary. If A's failure leaves a hole in the routing table of X, then X contacts the suggorate of A in an effort to identify a node to fill the hole. The details of the protocol can be found in the Tapestry papers.

In addition to repairing the routing mesh, the object location pointers also have to be adjusted. Objects rooted at the failed node may be inaccessible until the object is republished. The protocols for doing so essentially have to (i) maintain path availability, and (ii) optionally collect garbage/dangling pointers that would otherwise persist until the next soft-state refresh and timeout.

Overall, experiments have shown that Tapestry continues to perform well with high probability, despite dynamic node insertions and failures.

Complexity

- A search for an object is expected to take $(log_b 2^m)$ hops. However, the routing tables are optimized to identify nearest neighbor 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 neighbor set that is maintained for fault-tolerance.

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

18.7 Some other challenges in P2P system design

18.7.1 Fairness: a game theory application

P2P systems depend on all the nodes cooperating to store objects and allowing other nodes to download from them. However, nodes tend to be selfish in nature; thus there is a tendancy to download files without reciprocating by allowing others to download the locally available files. This behavior, termed as *leaching* or *free-riding*, leads to a degradation of the overall P2P system performance. Hence, penalties and incentives should be built in the system to encourage sharing and maximize the benefit to all nodes.

We now examine the classical problem, termed the *prisoners' dilemma*, from game theory, that has some useful lessons on how selfish agents might cooperate. This problem is an example of a non-zero-sum-game.

In the 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 prsioner remains silent, the betrayer gets freed and the silent accomplice gets 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 offence.

Rational selfish behavior dictates that both A and B would betray the other. This is not a Pareto-optimal solution, where a Pareto-optimal solution is one in which the overall good of all the participants is maximized. In the above example, both A and B staying silent results in a Pareto-optimal solution. The dilemma is that this is not considered the rational behavior of choice.

In the iterative prisoners' dilemma, the game is played multiple times, until an "equilibrium" is reached. Each player retains memory of the last move of both players (in more general versions, the memory extends to several past moves). After trying out various strategies, both players should converge to the ideal optimal solution of staying silent. This is Pareto-optimal.

The commonly accepted view is that the *tit-for-tat* strategy, described next, is the best for winning such a game. In the first step, a prisoner cooperates, and in each subsequent step, he reciprocates the action taken by the other party in the immediately preceding step.

The BitTorrent P2P system [11] has adopted the tit-for-tat strategy in deciding whether to allow a download of a file in solving the leaching problem. Here, cooperation is analogous to allowing others to upload local files, and betrayal is analogous to not allowing others to upload. The term *choking* 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 unchoking* is periodically done to unchoke peers that have been choked. This optimistic action roughly corresponds to the re-initiation of the game with the previously choked peer after some time epoch has elapsed.

18.7.2 Trust or reputation management

Various incentive-based economic mechanisms to ensure maximum cooperation among the selfish peers inherently depend on the notion of trust. In a P2P environment where the peer population is highly transient, there is also a need to have trust in the quality of data being downloaded. These requirements have lead to the area of trust and trust management in P2P systems [1, 18, 19]. As no node has a complete view of the other downloads in the P2P system, it may have to contact other nodes to evaluate the trust in particular offerers from which it could download some file. 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, and how to minimize the cost of the trust management protocols.

18.8 Tradeoffs between table storage and route lengths

18.8.1 Unifying DHT protocols

Chord, CAN, and Tapestry are three well-known representative protocols for managing structured P2P overlays. Despite their seeming differences, Xu *et al.* [34] showed that the routing function they perform can be expressed in a uniform way by generalizing the function of classless interdomain domain routing (CIDR) used by the IP protocol. We assume that all identifiers are in the common address space. We also assume modulo arithmetic.

Routing rule

The next-hop routing to node with identifier $dest$ from the current node with identifier id is as follows.

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 $R(x)$.

Clearly, we must have that for distinct i and j, $S_{id,i} \cap S_{id,j} = \emptyset$ and $J_{id,i} \neq J_{id,j}$. Further, $\cup_{1\leq i\leq s} S_{id,i}$ contains all the keys not stored by node id. When $S_{id,i}$ and $J_{id,i}$ are independent of id, as is the case for CAN, Chord, and Tapestry, the subscript id can be deleted.

- **Chord** if $dest - id \in S_i = [2^{i-1}, 2^i)$ then node id routes to node $id + J_i$, where $J_i = 2^{i-1}$.

  This corresponds to looking up the ith entry in the finger table, as described in Section 18.4.3.
- **CAN** The greedy routing function for CAN was given in Section 18.5.3. Here we assume a simple uniform distribution of nodes in the address space, $x^d = n$, and that nodes are numbered by an integer in base x, where x is the number of nodes in each dimension. Routing is assumed to be done dimension by dimension (rather than using greedy routing). Wraparound routing is assumed in each dimension. Then, for each dimension i, the following holds: if $dest$ and id differ in dimension i, route to i's neighbor in that dimension. Formally,

  If $dest - id \in (S_i =)[x^{i-1}, x^i)$ then route to $id + J_i$, where $J_i + id$ is a neighbor node in dimension in $i - 1$ and $J_i = kx^{i-1}$ for some $k \leq x$.
- **Tapestry** Let $x = log_b n$, $lvl = 1, \ldots, x$ and $j \in 0, \ldots, b-1$. After deleting the longest common prefix between id and $dest$, $prefix(dest, lvl-1)$, from $dest$, we have $suffix(dest, x - lvl + 1)$. The routing function was described in Section 18.6.2.

  If $suffix(dest, x - lvl + 1) \in S_{(lvl-1)\cdot b+j} = [j \cdot b^{x-lvl+1}, (j+1)\cdot b^{x-lvl+1})$ then node id routes to node $prefix(id, lvl-1) \circ suffix(J_{(lvl-1)\cdot b+j}, x - lvl + 1)$, where $J_{(lvl-1)\cdot b+j} \in [j \cdot b^{x-lvl+1}, (j+1)\cdot b^{x-lvl+1})$.

These routing relationships are summarized in Table 18.4.

Table 18.4 Comparison of representative P2P overlays. d is the number of dimensions in CAN. b is the base in Tapestry [34].

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}	$suffix(J_{(lvl-1) \cdot b+j}, x - lvl + 1)$

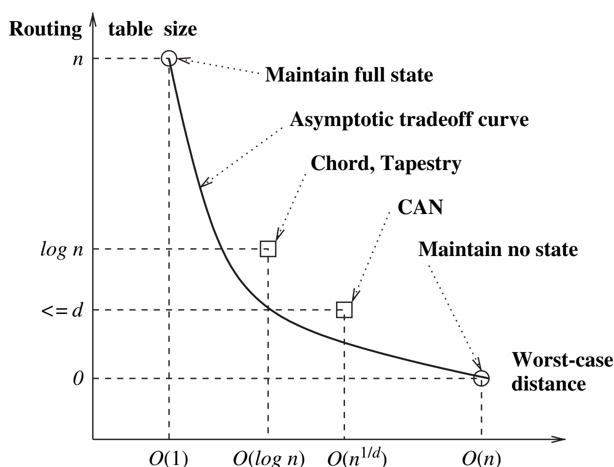

Figure 18.10 Fundamental asymptotic tradeoffs between router table size and network diameter [34].

18.8.2 Bounds on DHT storage and routing distance

Based on Table 18.4, the router table size and network diameter are represented in Figure 18.10. A fundamental question is whether the asymptotic bounds on (routing table size, network diameter as determined by the maximum number of hops) are $(log_2 n, \Omega(log_2 n))$ as for Chord and Tapestry, and $(d, \Omega(n^{1/d}))$ as for CAN. Xu *et al.* [34] used the following definitions to answer this:

- A routing algorithm is *weakly uniform* if for any nodes id and id', the jump sizes $J_{id,i} = J_{id',i}$. Thus, a weakly uniform algorithm requires the corresponding "jump sizes" for any index i to be the same for all nodes, irrespective of the node identifier.
- A routing algorithm is *strongly uniform* if it is weakly uniform and if for any nodes id and id', $S_{id,i} = S_{id',i}$. A strongly uniform algorithm requires all routing tables to also have the same corresponding sizes of the index ranges.
- A network is node-congestion-free (resp., edge-congestion-free) if all nodes (resp., edges) are handling the same average traffic. A network is congestion-free it it is node-congestion-free and edge-congestion-free.

Chord, CAN, and Tapestry are all congestion-free algorithms. A strongly uniform algorithm is node-congestion-free.

The following result has been shown by Xu *et al.* [34]:

- When the routing algorithms are weakly uniform, $\Omega(log_2 n)$ and $\Omega(n^{1/d})$ are the lower bounds on the diameter in networks with routing tables of sizes $O(log\, n)$ and d, respectively. As Chord, CAN, and Tapestry are strongly uniform, they achieve the asymptotic lower bounds in the tradeoff.

18.9 Graph structures of complex networks

P2P overlay graphs can have different structures. An intriguing question is to characterize the structure of overlay graphs. This question is a small part of a much wider challenge of how to characterize large networks that grow in a distributed manner without any coordination [4]. Such networks exist in the following:

- Computer science: the WWW graph (WWW), the Internet graph that models individual routers and interconnecting links (INTNET), and the autonomous systems (AS) graph in the Internet.
- Social networks (SOC), the phonecall graph (PHON), the movie actor collaboration graph (ACT), the author collaboration graph (AUTH), and citation networks (CITE).
- Linguistics: the word co-occurrence graph (WORDOCC), and the word synonym graph (WORDSYN).
- The power distribution grid (POWER).
- Nature: in protein folding (PROT), where nodes are proteins and an edge represents that the two proteins bind together, and in substrate graphs for various bacteria and micro-organisms (SUBSTRATE), where nodes are substrates and edges are chemical reactions in which substrates participate.

It is widely intuited that such complex graphs must display some organizational principles that are encoded in their topology in some subtle ways. This has driven research on a unification theory to determine a suitable model in which all such uncontrolled graphs are instantiations.

The first logical attempt to model large networks without any known design principles is to use random graphs. The random graph model, also known as the Erdos–Renyi (ER) model [14], assumes n nodes and a link between each pair of nodes with probability p, leading to $n(n-1)p/2$ edges. Many interesting mathematical properties have been shown for random graphs. However, the complex networks encountered in practice are not entirely random, and show some, somewhat intangible, organizational principles.

Three ideas have received much investigative attention in recent times [4]:

- **Small world networks** Even in very large networks, the path length between any pair of nodes is relatively small. This principle of a "small world" was popularized by sociologist Stanley Milgram by the "six degrees of separation" uncovered between any two people [24].

  As the average distance between any pair of nodes in the ER model grows logarithmically with n, the ER graphs are small worlds.
- **Clustering** Social networks are characterized by cliques. The degree of cliques in a graph can be measured by various clustering coefficients, such as the following. Consider a node i having k_i out-edges. Let l_i be the actual number of edges among the k_i nearest neighbors of i. If these k_i nearest neighbors were in a clique, they would have $k_i.(k_i - 1)/2$ edges among them. The clustering coefficient for node i is $C_i = 2l_i/(k_i(k_i - 1))$. The network-wide clustering coefficient is the average of all C_is, for all nodes i in the network.

  The random graph model has a clustering coefficient of exactly p. As most real networks have a much larger clustering coefficient, this random graph model (ER) is unsatisfactory.
- **Degree distributions** Let $P(k)$ be the probability that a randomly selected node has k incident edges. In many networks – such as INTER, AS, WWW, SUBST – $P(k) \sim k^{-\gamma}$, 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 with a peak of $P(\langle k \rangle)$, where $\langle k \rangle$, which is a function of n, is the average degree in the graph. Thus, random graphs are not scale-free. While some real networks have an exponential tail, the actual form of $P(k)$ is still very different from that for a Poisson distribution.

Current empirical measurements show the following properties of some commonly occuring graphs:

WWW In-degree and out-degree distributions both follow power laws; it is a small world; and is a directed graph, but does show a high clustering coefficient.

INTNET Degree distributions follow power law; small world; shows clustering.

AS Degree distributions follow power law; small world; shows clustering.

ACT Degree distributions follow power law tail; small world (similar path length as ER); shows high clustering.

AUTH Degree distributions follow power law; small world; shows high clustering.

SUBSTRATE In-degree and out-degree distributions both follow power laws; small world; large clustering coefficient.

PROT Degree distribution has a power law with exponential cutoff.
PHON In-degree and out-degree distributions both follow power laws.
CITE In-degree follows power law, out-degree has an exponential tail.
WORDOCC Two-regime power-law degree distribution; small world; high clustering coefficient.
WORDSYN Power-law degree distribution; small world; high clustering coefficient.
POWER Degree distribution is exponential.

Efforts on developing models focus on random graphs to model random phenomena, small worlds to interpolate between random graphs and structured clustered lattices, and scale-free graphs to study network dynamics and network evolutions.

18.10 Internet graphs

18.10.1 Basic laws and their definitions

In this section, we consider some properties of the Internet, that demonstrate a power-law behavior as measured empirically. The power law informally implies that large occurrences are very rare, and the frequency of the occurrence increases as the size decreases. Examples pertaining to the Web are: the number of links to a page, the number of pages within a Web location, and the number of accesses to a Web page. We begin by taking the example of the popularity of Websites to illustrate the definitions of three related observed laws [2]: Zipf's law, the Pareto law, and the Power law:

- **Power law** $P[X = x] \sim x^{-a}$
  This law is stated as a probability distribution function (PDF). It says that the number of occurrences of events that equal x is an inverse power of x. Figure 18.11(a) and (b) show the typical Power law PDF plots on both

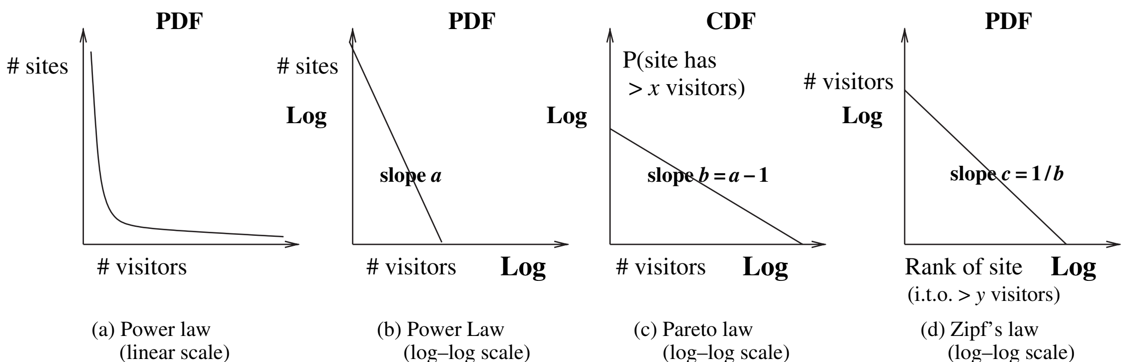

Figure 18.11 The popularity of Websites. (a) Power law showing the PDF using a linear scale. (b) Power law showing the PDF using a log–log scale. (c) Pareto law showing the CDF using a log–log scale. (d) Zipf's law using a log–log scale [2].

linear and log–log scales, respectively. In the log–log plot, the slope is a.

In our example, this corresponds to the number of sites that have exactly x visitors.

- **Pareto law** $P[X \geq x] \sim x^{-b} = x^{-(a-1)}$
  This law is stated as a cumulative distribution function (CDF). The number of occurrences larger than x is an inverse power of x. 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 $b+1=a$. Figure 18.11(c) shows the Pareto law CDF plot on a log–log scale. In the log–log plot, the slope is $b=a-1$.

  In our example, this corresponds to the number of sites that have at least x visitors.
- **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. Figure 18.11(d) shows the Zipf plot on a log–log scale. In the log–log plot, the slope is c, which, as we see below, is $1/b = 1/(a-1)$.

  The context initally used by Zipf was the frequency of occurrence of words in English, where the most frequently occurring word had rank 1. The Zipf law is widely occurring, e.g., both the magnitude of earthquakes and the populations of cities also follow this law. In our example, this corresponds to the number of visits to the rth most popular site.

Clearly, the Pareto law (CDF) and Power law (PDF) are related. Zipf's law $n \sim r^{-c}$, states that "the r-ranked object has $n = r^{-c}$ occurrences," and can be equivalently expressed as: "r objects (x-axis) have $n = r^{-c}$ (y-axis) or more occurrences." This becomes the same as the Pareto law's CDF after transposing the x and y axes, i.e., by restating as: "the number of occurrences larger than $n = r^{-c}$ (x-axis) happens for r objects (y-axis)."

From Zipf's law, $n = r^{-c}$, hence, $r = n^{-1/c}$. Hence, the Pareto exponent b is $1/c$. As $b = (a-1)$, where a is the Power law exponent, we see that $a = 1 + (1/c)$. Hence, the Zipf's law distribution also satisfies a Power law PDF.

18.10.2 Properties of the Internet

The Internet is a prime example of a complex entity that exhibits power-law behavior. Based on extensive empirical measurements, Siganos *et al.* [31] showed the following results:

- **Rank exponent/Zipf law** The nodes in the Internet graph are ranked in decreasing order of their degree. When the degree d_i is plotted as a function of the rank r_i on a log–log scale, the graph is like Figure 18.11(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 $m/n^{\mathcal{R}}$. Exercise 18.6 asks you to estimate the number of edges as a function of the rank exponent and the number of nodes.

- **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 function 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, $\mathcal{R} \sim (1/\mathcal{D})$, also as theoretically predicted. The imperfect match is attributed to imperfect measurements and approximations in curve-fitting. In practice, the CDF is preferred as it can be estimated with greater accuracy.
- **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 important 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 function of i is a straight line, with a slope of $\mathcal{E}$, the eigen-exponent. Thus, $\lambda_i \propto i^{\mathcal{E}}$. More intriguingly, when the eigenvalues and the degree are sorted in descending order, it is found that $\lambda_i = \sqrt{d_i}$, implying that $\mathcal{E} = \mathcal{D}/2$.

The following additional hypotheses have not been very vigorously tested and verified. Nevertheless, they offer insightful looks into the prevalance and use of power laws in complex uncontrolled entities such as the Internet. Two definitions are useful at this stage:

- $PN(h)$ is the number of pairs of nodes within h hops, counting self-pairs, and counting all other pairs twice due to the dual edge incidence.
- $NN(h)$, the neighborhood, is the expected number of nodes within h hops.

- **Hop-plot exponent**, $\mathcal{H}$ Experimental measurements have shown that $PN(h)$ follows a power law regime more closely, rather than the exponential regime as previously estimated. Thus, $PN(h) \propto h^{\mathcal{H}}$, where $\mathcal{H}$ is the slope of the log-log plot of $PN(h)$ as a function of h for $h \ll dia$. From the definition of $PN(h)$, observe that $PN(1) = n + 2l$, where l is the number of edges. Hence,

$$PN(h) = \begin{cases} (n+2l)h^{\mathcal{H}}, & \text{if } h \ll dia, \\ n^2, & \text{if } h \geq dia. \end{cases} \tag{18.9}$$

The hop-plot exponent is useful to estimate the effective diameter dia_{eff} of the network. Informally, any two nodes in the network are within dia_{eff}

hops of each other, with "high probability." When some destination node whose location is unknown needs to be reached, the use of hop-constrained broadcast is the standard solution. A large hop count takes too long, whereas a small hop count may not reach the entire network. If the hop count is set to dia_{eff}, then with high probability, the destination can be reached with just the right amount of overhead. Using n, $\mathcal{H}$, and the number of edges l, the effective diameter is defined as:

$$dia_{eff} = \left(\frac{n^2}{n+2l}\right)^{1/\mathcal{H}}.$$

This effective diameter is estimated as the abscissa of the intersection of the log–log hop-plot with slope $\mathcal{H}$ and the n^2 coverage that is expected within diameter hops.

Observe that the average size of the neighbourhood $NN(h) = (PN(h)/n) - 1$. Hence $NN(h) = ((n+2l)h^{\mathcal{H}}/n) - 1$. The $NN(h)$ is seen to be a more accurate estimate of the neighborhood than the traditional *average-degree estimate*, $NN'_d(h) = \overline{d}(\overline{d}-1)^{h-1}$. The $NN'_d(h)$ estimate assumes that the degree distribution is more uniform, and that each hop adds $\overline{d}-1$ new nodes per node at the boundary of the examined neighborhood. As the degree distribution is highly skewed, the traditional $NN'(h)$ metric is not accurate.

For all the cases above, the power law regime has so far been empirically validated. The exponent itself has been observed to change gradually over time as the networks evolve. The power law regime provides a good handle on predicting the future growth of the Internet, and building accurate graphs for simulations.

Classification of scale-free networks

Scale-free networks of different types – WWW, INTNET, AS, ACT, AUTH, SUBSTRATE, PROT, PHON, in-degree for CITE, and WORDSYN – have different degree exponents, typically ranging from 2 to 3. The quest to seek a more universal and common factor resulted in the analysis of another metric, called the "betweenness centrality" [15]. 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 betweeness centrality BC of node k, b_k, is $\sum_{i\neq j} g_k(i, j) = \sum_{i\neq j} |S_k(i, j)|/|S(i, j)|$. 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 varied network types. This interesting observation is under further study.

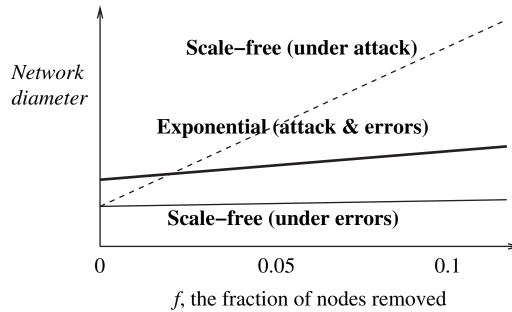

Figure 18.12 Impact of attacks and failures on the diameter of exponential networks and scale-free networks, from Albert *et al.* [5].

18.10.3 Error and attack tolerance of complex networks

Based on the node degree distribution $P(k)$, two broad classes of small world networks are the exponential networks and the scale-free networks. In exponential networks, such as the ER random graph model and the Watts–Strogatz small world model [33], $P(k)$ reaches a maximum at a $\overline{k}$ value and then $P(k)$ decreases exponentially per a Poisson distribution as k increases. In scale-free networks, such as the Web and the Internet, $P(k)$ decreases as per a power law, $P(k) \sim k^{-\gamma}$.

The following are two key differences that leads to different behavior of exponential networks and of scale-free networks, under errors and attacks: (i) nodes with a very high degree are statistically significant in scale-free networks, whereas they are close to an impossibility in exponential networks; (ii) in an exponential network, all nodes have about the same number of links, whereas in a scale-free network, some nodes have many links and the majority of the nodes have a small number of links.

Errors are simulated by removing nodes at random. Attacks are simulated by removing the nodes with highest degree. Their impact is measured on network diameter and network partitioning [5].

Impact on network diameter

Figure 18.12 is used to descibe the impact on the diameter. The graph shows only the relative trends, as empirically verified by simulations for many large networks, including the Web and Internet. Any numbers simply in the graph convey an approximate order of magnitude for the particular networks studied by Albert *et al.* [5].

- **Errors** In an exponential network, as all nodes have about the same degree, the removal of any node has approximately the same amount of small impact in terms of decrease in connectivity. The network diameter increases gradually. The diameter of scale-free networks remains almost same under errors, as nodes that are removed have small degree with very high probability and are very unlikely to alter the lengths of the paths among other nodes.

- **Attacks** As nodes in an exponential network have about the same degree, the network behaves similarly under attack as under errors. Under attack, the diameter of scale-free networks increases dramatically, as the few nodes with highest connectivity are removed, thereby greatly reducing the connectivity of the entire network.

Impact on network partitioning

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.

- **Exponential networks** In Figure 18.13, as f, the fraction of nodes removed is increased, S_{others} increases from 1 to around 2 for some threshold fraction $f_{threshold}$. This implies that for very small f, where $S_{others} \sim 1$, single nodes break off. As f increases, several small but larger partitions set in, leading to a peak of S_{others} at $f_{threshold}$. For $f > f_{threshold}$, S_{others} reduces back to 1, as the isolated clusters (fragments) in the network further disintegrate. In terms of S_{max}, as f is varied from 0 to $f_{threshold}$, S_{max} decreases from 1 to a low value as small (mostly single-node) partitions break off. As $f_{threshold}$ is approached, the main cluster disintegrates, leading to S_{max} tending to 0. As f is increased beyond $f_{threshold}$, S_{max} remains near 0.

  The impact of attacks on network partitioning is the same as the impact of errors, for the same reasoning given for the analysis on the diameter.
- **Scale-free networks** In Figure 18.14, when nodes are randomly removed, S_{max} decreases from 1 very gradually. Also, S_{others} remains steady at 1, indicating that singleton nodes get removed from the main network. There is no threshold $f_{threshold}$ observed, even for high values of f, such as 0.5 error rate.

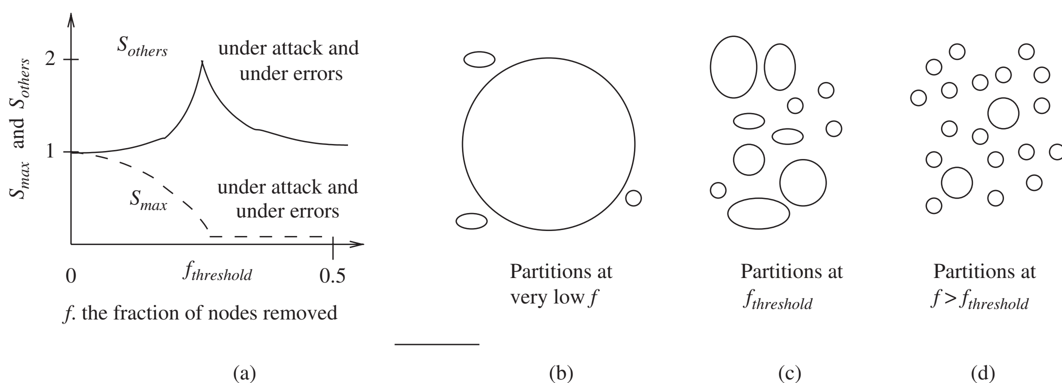

Figure 18.13 Impact of errors and attacks on cluster size of exponential networks, from Albert *et al.* [5]. (a) Graphical trend. (b) Pictoral cluster sizes for low f, i.e., $f \ll f_{threshold}$. (c) Pictoral cluster sizes for $f \sim f_{threshold}$. (d) Pictoral cluster sizes for $f > f_{threshold}$. The pictoral trend in (b)–(d) is also exhibited by scale-free networks under attack, but for a lower value of $f_{threshold}$.

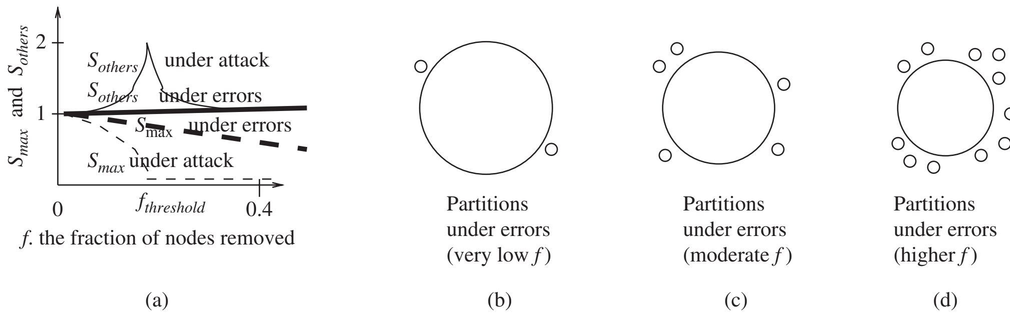

Figure 18.14 Impact of errors on cluster size of scale-free networks, from Albert *et al.* [5]. The pictoral impact of attacks on cluster sizes are similar to those in Figure 18.13. (a) Graphical trend. (b) Pictoral cluster sizes for low f under failure. (c) Pictoral cluster sizes for moderate f under failure. (d) Pictoral cluster sizes for high f under failure.

However, under attack, when the most connected nodes are removed, the behavior is similar to (but more acute than) that of the exponential network; see Figure 18.13. Thus, the threshold $f_{threshold}$ sets in at a lower value. This is because the impact of removing the highly connected nodes first causes disintegration to set in quickly.

18.11 Generalized random graph networks

Random graphs cannot capture the scale-free nature of real networks, which states that the node degree distribution follows a power law. The *generalized random graph model* uses the degree distribution as an input, but is random in all other respects. Thus, the constraint that the degree distribution must obey a power law is superimposed on an otherwise random selection of nodes to be connected by edges. These semi-random graphs can be analyzed for various properties of interest. Although a simple formal model for the clustering coefficient is not known, it has been observed that generalized random graphs have a random distribution of edges similar to the ER model, and hence the clustering coefficient will likely tend to zero as N increases.

18.12 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 tend to satisfy this property that clustering coefficients are independent of the network size. Figure 18.15(a) shows a one-dimensional

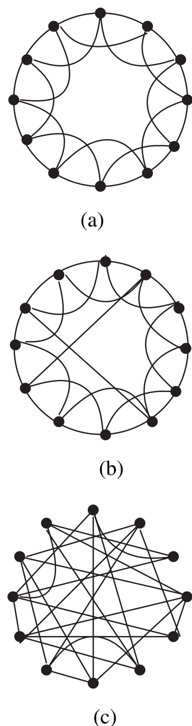

Figure 18.15. The Watts–Strogatz random rewiring procedure [4,33]. (a) Regular. (b) Small-world. (c) Random. The rewiring shown maintains the degree of each node.

lattice in which each node is connected to $k = 4$ closest nodes. The clustering coefficient is $C = \frac{3(k-2)}{4(k-1)}$.

The first model for small world graphs with high clustering coefficients and low path length is the Watts–Strogatz (WS) model [33]:

1. Define a ring lattice with n nodes and each node connected to k closest neighbors ($k/2$ 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 18.15(a). When $p = 1$, complete randomness, as in Figure 18.15(c).

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, it is now believed that the characteristic path length varies as:

$$len(n, p) \sim \frac{n^{1/d}}{k} f(pkn), \tag{18.10}$$

where the function f behaves as follows:

$$f(u) = \begin{cases} \text{constant}, & \text{if } u \ll 1, \\ ln(u)/u, & \text{if } u \gg 1. \end{cases} \tag{18.11}$$

The variable u 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."

18.13 Scale-free networks

Many real networks are scale-free, and even for those that are not scale-free, the degree distribution follows an exponential tail that is significantly different from that of the Poisson distribution. 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. Rather than modeling the network topology, it is better to model the network assembly and evolution process. Specifically:

Figure 18.16 The simple Barabasi–Albert model [7].

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, (where $m \leq m_0$), that link the new node to m different 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:

$$\prod(k_i) = \frac{k_i}{\sum_j (k_j)}. \tag{18.12}$$

- Rather than begin with a constant number of nodes n that are then randomly connected or rewired, real networks (e.g., WWW, INTERNET) exhibit *growth* by the addition of nodes and edges.
- Rather than assume that the probability of adding (or rewiring) an edge between two nodes is a constant, real networks exhibit the property of *preferential attachment*, where the probability of connecting to a node depends on the node degree.

The simple Barabasi–Albert model [7], which captures growth and preferential attachment, is described in Figure 18.16. After t time steps, there are $t + m_0$ nodes and mt edges. Numerically, it is verified that the degree distribution follows a power law with degree $= 3$, that is independent of the parameter m.

Two techniques to analyze the degree distribution of models are now described in the context of the BA model. The master-equation approach was introduced by Dogorotsev *et al.* [13] and the rate-equation approach was introduced by Krapivsky *et al.* [22].

18.13.1 Master-equation approach

Let $p(k, t_i, t)$ denote the probability that, at time t, a node i that was added at time t_i has degree k. When a new node with m edges is added to the graph, the degree of node i increases by one with probability $m \cdot \prod(k) = k/2t$. Hence, we have [4, 13]:

$$p(k, t_i, t+1) = \frac{k-1}{2t} \cdot p(k-1, t_i, t) - \left[1 - \frac{k}{2t}\right] \cdot p(k, t_i, t). \tag{18.13}$$

The first term is the probability that a node with $k-1$ degree gets a new edge; the second term is the probability that a node with degree k does not get a new edge. Based on this formulation, the degree distribution can be expressed as:

$$P(k) = limit_{t\to\infty} \sum_{t_i} p(k, t_i, t)/t. \tag{18.14}$$

From Eq. (18.13), it can be shown that:

$$P(k) = \begin{cases} \frac{k-1}{k+2} P(k-1), & \text{if } k \geq m+1, \\ \frac{2}{m+2}, & \text{if } k = m. \end{cases} \quad (18.15)$$

This solves as:

$$P(k) = \frac{2m(m+1)}{k(k+1)(k+2)}. \quad (18.16)$$

18.13.2 Rate-equation approach

Let $n_k(t)$ be the average number of nodes having k edges at time t. When a new node is added, $n_k(t)$ changes as follows. New edges are added to some nodes with degree $k-1$, new edges are added to some nodes with degree k, and new nodes with m edges are added. These three changes affect $n_k(t)$ in the following manner:

$$\frac{dn_k}{dt} = m \cdot \left[\frac{(k-1) \cdot n_{k-1}(t)}{\sum_k kn_k(t)} - \frac{k \cdot n_k(t)}{\sum_k kn_k(t)} \right] + \delta_{k,m}. \quad (18.17)$$

By taking the asymptotic limit, $n_k(t) = t \cdot P(k)$, and $\sum_k kn_k(t) = 2mt$. This yields the same recursive Eq. (18.15) obtained using the master-equation approach.

18.14 Evolving networks

The BA algorithm in Figure 18.16 represents a basic model that cannot fully capture real network properties. For example, the BA model has a fixed exponent of 3 for the power law, independent of the parameter m. Real networks have an exponent that varies, typically between 1 and 3. Some real networks sometimes have exponential cutoffs that are not within the power law regime. The study of more general and flexible models that can accurately capture real networks has lead to several notable directions of investigation:

- **Preferential attachment** The BA model assumed that the probability $\prod(k)$ that a new node connects to a node i is proportional to the degree k_i. This implied that $\prod(k)$ is linearly proportional to k.

  It has been shown analytically that for *sublinear preferential attachment* as well as for *superlinear preferential attachment*, the scale-free nature of the network cannot be preserved.

  In real networks, there is a finite probability that a new node attaches to an isolated node, i.e., $\prod(0) \neq 0$ and $\prod(k) = C + k^{\alpha}$, where C denotes the *intial*

attractiveness. It can be seen that initial attractiveness changes the degree exponent but preserves the scale-free nature of the degree distribution.

- **Growth** The BA model assumed that the rate of addition of nodes and edges was uniform. Many real networks, such as INTNET, AS, WEB, SUBSTRATE, and WORDOCC, have the property that the number of edges increases faster than the number of nodes, implying an increase in the average degree as the number of nodes increases. It has been shown analytically that accelerated growth does not affect the power law nature although the exponent degree is altered.
- **Local events** Real networks undergo local (microscopic) changes to the topology, such as node addition and node deletion, edge addition and edge deletion. A popular model that explores the properties of such local events is the extended Barabasi–Albert model [3], shown in Figure 18.17.
- **Growth constraints** Real networks often have bounded capacity for the number of edges (e.g., connections at a router) or a finite lifetime for the nodes (as in social networks). In the electrical power distribution network which exhibits an exponential distribution, there are practical reasons why the node degree is bounded. In the actors network, which exhibits a power law with an exponential cutoff for large k, ageing limits the accrual of new edges. Thus, ageing and finite capacity need to explicitly captured in a good model for such networks.
- **Competition** Real-world networks exhibit competition, wherein some nodes can attract more edges (e.g., via advertising) at the cost of other nodes. This feature can be modeled by a fitness parameter. Similarly, a new node may inherit edges belonging to some other node or nodes (e.g., modifying a replica of a Web page). This needs to be explicitly modeled.
- **Induced preferential attachment** Various local-level mechanisms, such as the copying mechanism (copy edges of another node as in Web

Figure 18.17 The extended Barabasi–Albert model [3].

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

$$\prod(k_i) = \frac{k_i + 1}{\sum_j (k_j + 1)}. \tag{18.18}$$

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 Eq. (18.18).

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.

pages), and tracing selected walks (as in recursively following the citation trail in a citation network), need to be modeled because they implicitly introduce preferential attachment.

18.14.1 Extended Barabasi–Albert model

The extended BA model [3] is an example model for evolving networks.

Continuum theory analysis

In continuum theory, it is assumed that k_i changes continuously and the probability $\prod(k_i)$ then represents the rate at which k_i changes. Each of the three possible events in a sequential step can affect the rate at which k_i changes as follows [3]:

1. With probability p, m new links are added. For each link, one end is randomly chosen, leading to a change in k_i of pm/n. For each link, the second end attaches preferentially, leading to a change in k_i of $pm \cdot \frac{(k_i+1)}{\sum_j (k_j+1)}$. Hence,

$$\frac{dk_i}{dt} = pm\frac{1}{n} + pm\frac{k_i+1}{\sum_j (k_j+1)}. \tag{18.19}$$

2. With probability q, m existing links are rewired. For each rewired link, a randomly chosen node loses one incident edge, which then attaches preferentially. Thus, the impact on k_i is:

$$\frac{dk_i}{dt} = -qm\frac{1}{n} + qm\frac{k_i+1}{\sum_j (k_j+1)}. \tag{18.20}$$

3. With probability $(1-p-q)$, a new node is added with m links. Each of the m links connects preferentially, thus:

$$\frac{dk_i}{dt} = (1-p-q)m\frac{k_i+1}{\sum_j (k_j+1)}. \tag{18.21}$$

Summing the three effects, we have:

$$\frac{dk_i}{dt} = (p-q)m\frac{1}{n} + m\frac{k_i+1}{\sum_j (k_j+1)}. \tag{18.22}$$

As the system size and topology varies with time, we have:

$$n(t) = m_0 + (1-p-q)t; \qquad \sum_j k_j = 2mt(1-q) - m. \tag{18.23}$$

As t increases, the constants m and m_0 can be deleted. Further, for a node added at t_i, we have that $k_i(t_i) = m$ (the initialization step). Exercise 18.8 asks you to show that the solution to Eq. (18.22) has the form

$$k_i(t) = [A(p,q,m) + m + 1]\left(\frac{t}{t_i}\right)^{1/B(p,q,m)} - A(p,q,m) - 1, \qquad (18.24)$$

$$A(p,q,m) = (p-q)\left(\frac{2m(1-q)}{1-p-q} + 1\right), \qquad B(p,q,m) = \frac{2m(1-q)+1-p-q}{m}. \qquad (18.25)$$

Based on further algebraic derivations, Albert and Barabasi [3] showed that:

$$P(k) \ \alpha \ [k + \kappa(p,q,m)]^{-\gamma(p,q,m)}, \text{ where } \kappa(p,q,m) = A(p,q,m) + 1 \text{ and}$$
$$\gamma(p,q,m) = B(p,q,m) + 1, \qquad (18.26)$$

Equation (18.26) is valid if, for a fixed p and m,

$$q < q_{max} = min(1-p, (1-p+m)/(1+2m)).$$

There are now two cases:

$q < q_{max}$: Eq. (18.26) is valid and the degree distribution is a power law and is scale-free.

$q > q_{max}$: Eq. (18.26) is invalid, and $P(k)$ can be shown to behave like an exponential distribution. The model now behaves like the ER and WS models.

This is similar to the behavior 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

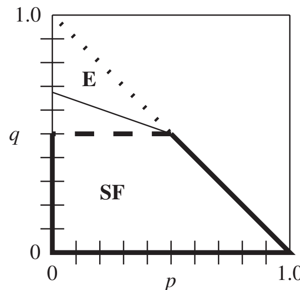

Figure 18.18 Phase diagram for the extended Barabasi–Albert model [3]. **SF** denotes the scale-free regime, which is enclosed by the thick border. **E** denotes the exponential regime that exists in the remainder of the lower diagonal region of the graph. The plain line shows the boundary for $m = 1$, having a y-axis intercept at 0.67.

and the exponential regime are marked in the graph in Figure 18.18. 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 $m \to 0$.

18.15 Chapter summary

Peer-to-peer (P2P) networks allow equal participation and resource sharing among the users. This chapter first analyzed the different types of P2P networks. Unstructured P2P networks are like Gnutella and BitTorrent. We studied different search mechanisms – flooding, constrained flooding, and blind search – for such unstructured networks. We also examined some data replication strategies, and their impact on the search performance. The chapter then studied three classical structured P2P networks – Chord, CAN, Tapestry – all of which use the distributed hash table concept in their implementations. Although all the three mechanisms differ, they are similar in that they represent different tradeoffs in search efficiency, i.e., path length, and the amount of local storage for implementing the hash tables. The spectrum of P2P networks from unstructured to structured offer a wide range of tradeoffs for user requirements. The chapter also examined issues such as fairness and trust management. These issues are important because, in the P2P environment where there is no control authority, the system must be able to autonomously alllow for fairness.

The Internet, AS-AS level internets, and Web (WWW) overlays exhibit some interesting properties about how they grow and evolve. Many network overlays outside of computer science also exhibit the same properties. The chapter studied several properties of the Internet and Web graphs. Then, in a more general setting, the chapter examined random networks, small-world networks, node degree distributions, scale-free networks, and the impact of error and attack tolerance on such networks. Networks grow in an uncontrolled fashion, yet there seems to be some underlying basis for such growth. Of the several proposals to model the growth of networks, we studied the Barabasi–Albert model, which appears to be promising in its applicability to not just computer science networks, but also to networks in other disciplines and natural phenomena.

18.16 Exercises

Exercise 18.1 (Replication) Derive the values of average search size A, A_i, and utilization u_i for square-root replication. The derived answers should match the entries in Table 18.3.

Exercise 18.2 (Fault-tolerance in Chord) Adapt the code in Algorithm 18.3 so that the nodes manage a successor list of α successors, rather than a single successor.

Exercise 18.3 (Chord) In the Chord protocol, assume that the successor list at each node has $\alpha = \Omega(log\, n)$ nodes. Show the following:

1. If a Chord ring is initially stable, and if the probability of subsequent failure of each node is 0.5, then *Locate_Successor* returns the closest functional successor node to the key being searched with high probability.
2. If a Chord ring is initially stable, and if the probability of subsequent failure of each node is 0.5, it takes $O(log\, n)$ average-case time for *Locate_Successor* to complete.

Exercise 18.4 (CAN) Compute the time and message complexity of the distributed region reassignment protocol that is run periodically by the CAN protocol.

Exercise 18.5 (CAN) Identify all the changes to the base CAN protocol to accommodate the optimization of overloading coordinate regions, discussed in Section 18.5.5.

Exercise 18.6 (Power law in the Internet [31]) Show that the number of edges l in the Internet graph that obeys the power law for the rank exponent is given as follows. Let the graph have n nodes and rank exponent $\mathcal{R}$. Then:

$$l \sim \frac{1}{2(\mathcal{R}+1)}(1-\frac{1}{n^{\mathcal{R}+1}})n.$$

Exercise 18.7 Show that Eq. (18.15) using the master-equation approach for the degree distribution in the extended BA model can be solved as Eq. (18.16).

Exercise 18.8 Show that the solution to Eq. (18.22) for the degree distribution in the extended BA model using continuum theory analysis is given by Eq. (18.25).

18.17 Notes on references

The introduction is based on the survey by Risson and Moors [29] and Androutsellis-Theotokis and Spinellis [6]. The discussion on replication and search in unstructured networks is based on Cohen and Shenker [12], and on Lv *et al.* [23], respectively. Gnutella [16, 17], Napster [25], and Freenet [10] are widely implemented commercial P2P protocols. The Chord protocol was proposed by Stoica *et al.* [32]. The content addressible network (CAN) was proposed by Ratnasamy *et al.* [27]. The design of Tapestry [20, 21, 35, 36] and the related Pastry [30] overlay was based on the ideas of Plaxton trees proposed by Plaxton *et al.* [26]. Tapestry built on the Plaxton trees by providing better fault-tolerance and resilience in the face of node joins and departures. The discussion on fundamental tradeoffs between routing table size and network diameter is based on Xu *et al.* [34] and Ratnasamy *et al.* [28]. The BitTorrent system was initially proposed by Cohen [11]. The discussion of trust management is based on Gupta *et al.* [18, 19] and Aberer and Despotovic [1].

The discussion on the graph structures of complex networks is structured and based on the excellent survey by Albert and Barabasi [4]. The discussion on power laws and Zipf's law is taken from the tutorial by Adamic [2]. The power laws for the Internet

were discovered by Siganos and the Faloutsos brothers [31]. The discussion on the betweenness centrality metric for graphs is based on the work by Goh *et al.* [15]. The random graphs model was proposed and analyzed by Erdos and Renyi [14]. Further results on the properties on random graphs were given by Bollobas [8, 9]. The small worlds model was proposed by Watts and Strogatz [33]. The extended Barabasi–Albert model for graph evolution was given by Albert and Barabasi [3]. The analysis of error and attack tolerance on exponential networks and on scale-free networks was done by Albert *et al.* [5].

References

[1] K. Aberer and Z. Despotovic, Managing trust in a peer-to-peer information system, *Proceedings of the 10th International Conference on Information and Knowledge Management*, Atlanta, Georgia, USA, November 2001, 310–317.

[2] L. Adamic, *Zipf, Power-Laws, and Pareto – A Ranking Tutorial*, available online at: www.hpl.hp.com/research/idl/papers/ranking/ranking.html.

[3] R. Albert and A.-L. Barabasi, Topology of evolving networks: local events and universality, *Physical Review Letters*, **85**(24), 2000, 5234–5237.

[4] R. Albert and A.-L. Barabasi, Statistical mechanics of complex networks, *Review of Modern Physics*, **74**(1), 2002, 47–97.

[5] R. Albert, H. Jeong, and A. Barabasi, Error and attack tolerance of complex networks, *Nature*, **406**, 2000, 378–381.

[6] S. Androutsellis-Theotokis and D. Spinellis, A survey of peer-to-peer content distribution technologies, *ACM Computing Surveys*, **36**(4), 2004, 335–371.

[7] A.-L. Barabasi and R. Albert, Emergence of scaling in random networks, *Science*, 286, 1999, 509–512.

[8] B. Bollobas, Degree sequences of random graphs, *Discrete Math*, **33**, 1981, 1–9.

[9] B. Bollobas, *Random Graphs*, London, Academic Press, 1985.

[10] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong, Freenet: a distributed anonymous information storage and retrieval system, *Workshop on Design Issues in Anonymity and Unobservability*, Berkeley, CA, July 2000, 46–66.

[11] B. Cohen, *Incentives Build Robustness in BitTorrent*, available online at: www.bittorrent.com/bittorrentecon.pdf.

[12] E. Cohen and S. Shenker, Replication strategies in unstructured peer-to-peer networks, *ACM SIGCOMM*, 2002, 177–190.

[13] S. Dogorotsev, J. Mendes, and A. Samukhin, Structure of growing networks: exact solution of the Barabasi–Albert model, *Physical Review Letters*, **85**, 2000, 4633–4636.

[14] P. Erdos and A. Renyi, Random graphs. **6**, 1959, 290–.

[15] K. Goh, E. Oh, H. Jeong, B. Kahng, and D. Kim, Classification of scale-free networks, *Proceedings of the National Academy of Sciences*, 2002.

[16] Gnutella, `www.gnutella.com/`.

[17] The Gnutella protocol specification, available online at: `www9.limewire.com/developer/gnutella_protocol_0.4.pdf`.

[18] M. Gupta, P. Judge, and M. Ammar, A reputation system for peer-to-peer networks, *Proceedings of the 13th International Workshop on Network and Operating Systems Support for Digital Audio and Video*, Monterey, CA, June 2003, 144–152.

[19] M. Gupta, M. H. Ammar, and M. Ahamad, Trade-offs between reliability and overheads in peer-to-peer reputation tracking, *Computer Networks*, **50**(4), 2006, 501–522.

[20] K. Hildrum, J. Kubiatowicz, S. Rao, and B. Y. Zhao, Distributed object location in a dynamic network, *Proceedings of ACM SPAA 2002*, 41–52.

[21] K. Hildrum, J. Kubiatowicz, S. Rao, and B. Y. Zhao, Distributed object location in a dynamic network, *Theory of Computing Systems*, **37**, 2004, 405–440.

[22] P. Krapivsky, S. Redner, and F. Leyvraz, Connectivity of growing random networks, *Physical Review Letters*, **85**, 2000, 4629–4632.

[23] Q. Lv, P. Cao, E. Cohen, K. Li, and S. Shenker, Search and replication in unstructured peer-to-peer networks, *International Conference on Supercomputing*, 2002, 84–95.

[24] S. Milgram, The small world problem, *Psychology Today*, **1**(2), 1967, 60–67.

[25] Napster, `www.napster.com/`.

[26] C. G. Plaxton, R. Rajaraman, and A. W. Richa, Accessing nearby copies of replicated objects in a distributed environment, *Proceedings of ACM SPAA 1997*, 311–320.

[27] S. Ratnasamy, P. Francis, M. Handley, R.M. Karp, and S. Shenker, A scalable content-addressable network, *Proceedings of ACM SIGCOMM 2001*, 161–172.

[28] S. Ratnasamy, I. Stoica, and S. Shenker, Routing algorithms for DHTs: some open questions, *Proceedings of IPTPS 2002*, 45–52.

[29] J. Risson and T. Moors, Survey of research towards robust peer-to-peer networks: search methods, *Computer Networks*, **50**(17), 2006, 3485–3521.

[30] A. Rowstron and P. Druschel, Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems, *Proceedings of the IFIP/ACM Middleware 2001*, Heidelberg, Germany, November 2001, 329–350.

[31] G. Siganos, M. Faloutsos, P. Faloutsos, and C. Faloutsos, Power laws and the AS-level internet topology, *IEEE/ACM Transactions on Networking*, **11**(4), 2003, 514–524.

[32] I. Stoica, R. Morris, D. Liben-Nowell, D. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan, Chord: a scalable peer-to-peer lookup service for internet applications, *IEEE Transactions on Networking*, **11**(1), 2003, 17–31.

[33] D. J. Watts and S. H. Strogatz, Collective dynamics of “Small World” networks, *Nature*, No. 393, 1998, 440–442.

[34] J. Xu, A. Kumar, and X. Yu, On the fundamental tradeoffs between routing table size and network diameter in peer-to-peer networks, *IEEE Journal on Selected Areas in Communications*, **22**(1), 2004, 151–163.

[35] B. Y. Zhao, L. Huang, J. Stribling, S. Rhea, A. Joseph, and J. Kubiatowicz, Tapestry: a resilient global-scale overlay for service deployment, *IEEE Journal on Selected Areas in Communications*, **22**(1), 2004, 41–53.

[36] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph, *Tapestry: An Infrastructure for Fault-Resilient Wide-Area Location and Routing*, Technical Report UC Berkeley, CSD-01-1141, University of California at Berkeley, Berkeley, CA, 2001.