is the lightest edge that connects *any* two super-fragments, it must hold that e is exactly the blue edge of $\mathcal{F}'$. Thus, whenever an edge is added, it is an MST edge. □

Theorem 13.5. *Algorithm 47 computes an MST in time $O(\log \log n)$.*

Proof. Let β_k denote the size of the smallest fragment after phase k of Algorithm 47. We first show that every fragment merges with at least β_k other fragments in each phase. Since the size of each fragment after phase k is at least β_k by definition, we get that the size of each fragment after phase $k+1$ is at least $\beta_k(\beta_k+1)$. Assume that a fragment F, consisting of at least β_k nodes, does not merge with β_k other fragments in phase $k+1$ for any $k \geq 0$. Note that F cannot be safe because being safe implies that there is at least one edge in E' that has not been considered yet and that is the blue edge of F. Hence, the phase cannot be completed in this case. On the other hand, if F is not safe, then at least one of its sub-fragments has used up all its β_k edges to other fragments. However, such an edge is either used to merge two fragments or it must have been dropped because the two fragments already belong to the same fragment because another edge connected them (in the same phase). In either case, we get that any fragment, and in particular F, must merge with at least β_k other fragments.

Given that the minimum fragment size grows (quickly) in each phase and that only edges belonging to the MST are added according to Lemma 13.4, we conclude that the algorithm correctly computes the MST. The fact that

$$\beta_{k+1} \geq \beta_k(\beta_k+1)$$

implies that $\beta_k \geq 2^{2^{k-1}}$ for any $k \geq 1$. Therefore after $1+\log_2 \log_2 n$ phases, the minimum fragment size is n and thus all nodes are in the same fragment. □

Remarks:

- It is not known whether the $O(\log \log n)$ time complexity of Algorithm 47 is optimal. In fact, no lower bounds are known for the MST construction on graphs of diameter 1 and 2.
- Algorithm 47 makes use of the fact that it is possible to send different messages to different nodes. If we assume that every node always has to send the same message to all other nodes, Algorithm 46 is the best that is known. Also for this simpler case, no lower bound is known.

Chapter 14

Peer-to-Peer Computing

"Indeed, I believe that virtually *every* important aspect of programming arises somewhere in the context of [sorting and] searching!"

– Donald E. Knuth, The Art of Computer Programming

14.1 Introduction

Unfortunately, the term *peer-to-peer* (P2P) is ambiguous, used in a variety of different contexts, such as:

- In popular media coverage, P2P is often synonymous to software or protocols that allow users to "share" files, often of dubious origin. In the early days, P2P users mostly shared music, pictures, and software; nowadays books, movies or tv shows have caught on. P2P file sharing is immensely popular, currently at least half of the total Internet traffic is due to P2P!
- In academia, the term P2P is used mostly in two ways. A narrow view essentially defines P2P as the "theory behind file sharing protocols". In other words, how do Internet hosts need to be organized in order to deliver a search engine to find (file sharing) content efficiently? A popular term is "distributed hash table" (DHT), a distributed data structure that implements such a content search engine. A DHT should support at least a search (for a key) and an insert (key, object) operation. A DHT has many applications beyond file sharing, e.g., the Internet domain name system (DNS).
- A broader view generalizes P2P beyond file sharing: Indeed, there is a growing number of applications operating outside the juridical gray area, e.g., P2P Internet telephony à la Skype, P2P mass player games on video consoles connected to the Internet, P2P live video streaming as in Zattoo or StreamForge, or P2P social storage such as Wuala. So, again, what is P2P?! Still not an easy question... Trying to account for the new applications beyond file sharing, one might define P2P as a large-scale distributed system that operates without a central server bottleneck. However, with

this definition almost everything we learn in this course is P2P! Moreover, according to this definition early-day file sharing applications such as Napster (1999) that essentially made the term P2P popular would not be P2P! On the other hand, the plain old telephone system or the world wide web do fit the P2P definition...

- From a different viewpoint, the term P2P may also be synonymous for privacy protection, as various P2P systems such as Freenet allow publishers of information to remain anonymous and uncensored. (Studies show that these freedom-of-speech P2P networks do not feature a lot of content against oppressive governments; indeed the majority of text documents seem to be about illicit drugs, not to speak about the type of content in audio or video files.)

In other words, we cannot hope for a single well-fitting definition of P2P, as some of them even contradict. In the following we mostly employ the academic viewpoints (second and third definition above). In this context, it is generally believed that P2P will have an influence on the future of the Internet. The P2P paradigm promises to give better scalability, availability, reliability, fairness, incentives, privacy, and security, just about everything researchers expect from a future Internet architecture. As such it is not surprising that new "clean slate" Internet architecture proposals often revolve around P2P concepts.

One might naively assume that for instance scalability is not an issue in today's Internet, as even most popular web pages are generally highly available. However, this is not really because of our well-designed Internet architecture, but rather due to the help of so-called overlay networks: The Google website for instance manages to respond so reliably and quickly because Google maintains a large distributed infrastructure, essentially a P2P system. Similarly companies like Akamai sell "P2P functionality" to their customers to make today's user experience possible in the first place. Quite possibly today's P2P applications are just testbeds for tomorrow's Internet architecture.

14.2 Architecture Variants

Several P2P architectures are known:

- Client/Server goes P2P: Even though Napster is known to the be first P2P system (1999), by today's standards its architecture would not deserve the label P2P anymore. Napster clients accessed a central server that managed all the information of the shared files, i.e., which file was to be found on which client. Only the downloading process itself was between clients ("peers") directly, hence peer-to-peer. In the early days of Napster the load of the server was relatively small, so the simple Napster architecture made a lot of sense. Later on, it became clear that the server would eventually be a bottleneck, and more so an attractive target for an attack. Indeed, eventually a judge ruled the server to be shut down, in other words, he conducted a juridical denial of service attack.

- Unstructured P2P: The Gnutella protocol is the anti-thesis of Napster, as it is a fully decentralized system, with no single entity having a global picture. Instead each peer would connect to a random sample of other peers, constantly changing the neighbors of this virtual overlay network by exchanging neighbors with neighbors of neighbors. (In such a system it is part of the challenge to find a decentralized way to even discover a first neighbor; this is known as the bootstrap problem. To solve it, usually some random peers of a list of well-known peers are contacted first.) When searching for a file, the request was being flooded in the network (Algorithm 11 in Chapter 3). Indeed, since users often turn off their client once they downloaded their content there usually is a lot of *churn* (peers joining and leaving at high rates) in a P2P system, so selecting the right "random" neighbors is an interesting research problem by itself. However, unstructured P2P architectures such as Gnutella have a major disadvantage, namely that each search will cost m messages, m being the number of virtual edges in the architecture. In other words, such an unstructured P2P architecture will not scale.

- Hybrid P2P: The synthesis of client/server architectures such as Napster and unstructured architectures such as Gnutella are hybrid architectures. Some powerful peers are promoted to so-called superpeers (or, similarly, trackers). The set of superpeers may change over time, and taking down a fraction of superpeers will not harm the system. Search requests are handled on the superpeer level, resulting in much less messages than in flat/homogeneous unstructured systems. Essentially the superpeers together provide a more fault-tolerant version of the Napster server, all regular peers connect to a superpeer. As of today, almost all popular P2P systems have such a hybrid architecture, carefully trading off reliability and efficiency, but essentially not using any fancy algorithms and techniques.

- Structured P2P: Inspired by the early success of Napster, the academic world started to look into the question of efficient file sharing. Indeed, even earlier, in 1997, Plaxton, Rajaraman, and Richa proposed a hypercubic architecture for P2P systems. This was a blueprint for many so-called structured P2P architecture proposals, such as Chord, CAN, Pastry, Tapestry, Viceroy, Kademlia, Koorde, SkipGraph, SkipNet, etc. In practice structured P2P architectures are not yet popular, apart from the Kad (from Kademlia) architecture which comes for free with the eMule client. Indeed, also the Plaxton et al. paper was standing on the shoulders of giants. Some of its eminent precursors are:
  - Research on linear and consistent hashing, e.g., the paper "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web" by Karger et al. (co-authored also by the late Daniel Lewin from Akamai), 1997.
  - Research on locating shared objects, e.g., the papers "Sparse Partitions" or "Concurrent Online Tracking of Mobile Users" by Awerbuch and Peleg, 1990 and 1991.
  - Work on so-called compact routing: The idea is to construct routing tables such that there is a trade-off between memory (size of routing tables) and stretch (quality of routes), e.g., "A trade-off between space and efficiency for routing tables" by Peleg and Upfal, 1988.

– ...and even earlier: hypercubic networks, see next section!

14.3 Hypercubic Networks

(Thanks to Christian Scheideler, TUM, for the pictures in this section.)

In this section we will introduce some popular families of network topologies. These topologies are used in countless application domains, e.g., in classic parallel computers or telecommunication networks, or more recently (as said above) in P2P computing. Similarly to Chapter 4 we employ an All-to-All communication model, i.e., each node can set up direct communication links to arbitrary other nodes. Such a virtual network is called an *overlay network*, or in this context, P2P architecture. In this section we present a few overlay topologies of general interest.

The most basic network topologies used in practice are trees, rings, grids or tori. Many other suggested networks are simply combinations or derivatives of these. The advantage of trees is that the routing is very easy: for every source-destination pair there is only one possible simple path. However, since the root of a tree is usually a severe bottleneck, so-called *fat trees* have been used. These trees have the property that every edge connecting a node v to its parent u has a capacity that is equal to all leaves of the subtree routed at v. See Figure 14.1 for an example.

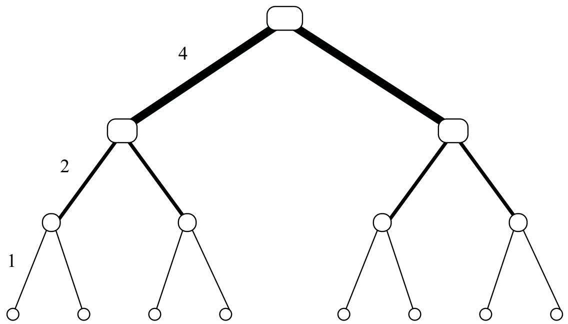

Figure 14.1: The structure of a fat tree.

Remarks:

- Fat trees belong to a family of networks that require edges of non-uniform capacity to be efficient. Easier to build are networks with edges of uniform capacity. This is usually the case for grids and tori. Unless explicitly mentioned, we will treat all edges in the following to be of capacity 1. In the following, $[x]$ means the set $\{0, \dots, x-1\}$.

Definition 14.1 (Torus, Mesh)**.** *Let $m, d \in \mathbb{N}$. The (m,d)-mesh $M(m,d)$ is a graph with node set $V = [m]^d$ and edge set*

$$E = \left\{ \{(a_1, \dots, a_d), (b_1, \dots, b_d)\} \mid a_i, b_i \in [m], \sum_{i=1}^{d} |a_i - b_i| = 1 \right\}.$$

The (m,d)-torus $T(m,d)$ is a graph that consists of an (m,d)-mesh and additionally wrap-around edges from nodes $(a_1, \dots, a_{i-1}, m, a_{i+1}, \dots, a_d)$ to nodes $(a_1, \dots, a_{i-1}, 1, a_{i+1}, \dots, a_d)$ for all $i \in \{1, \dots, d\}$ and all $a_j \in [m]$ with $j \neq i$. In other words, we take the expression $a_i - b_i$ in the sum modulo m prior to computing the absolute value. $M(m,1)$ is also called a line*, $T(m,1)$ a* cycle*, and $M(2,d) = T(2,d)$ a d-*dimensional* hypercube*. Figure 14.2 presents a linear array, a torus, and a hypercube.*

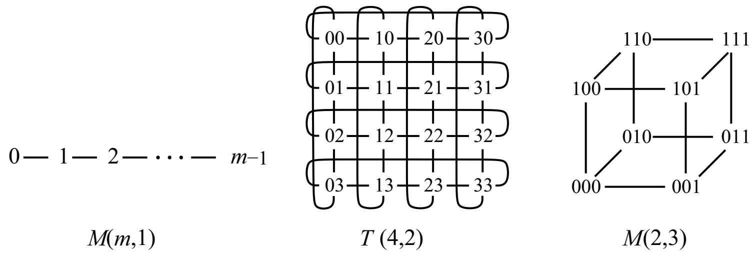

Figure 14.2: The structure of $M(m,1)$, $T(4,2)$, and $M(2,3)$.

Remarks:

- Routing on mesh, torus, and hypercube is trivial. On a d-dimensional hypercube, to get from a source bitstring s to a target bitstring d one only needs to fix each "wrong" bit, one at a time; in other words, if the source and the target differ by k bits, there are $k!$ routes with k hops.

- The hypercube can directly be used for a structured P2P architecture. It is trivial to construct a distributed hash table (DHT): We have n nodes, n for simplicity being a power of 2, i.e., $n = 2^d$. As in the hypercube, each node gets a unique d-bit ID, and each node connects to d other nodes, i.e., the nodes that have IDs differing in exactly one bit. Now we use a globally known hash function f, mapping file names to long bit strings; SHA-1 is popular in practice, providing 160 bits. Let f_d denote the first d bits (prefix) of the bitstring produced by f. If a node is searching for file name X, it routes a request message $f(X)$ to node $f_d(X)$. Clearly, node $f_d(X)$ can only answer this request if all files with hash prefix $f_d(X)$ have been previously registered at node $f_d(X)$.

- There are a few issues which need to be addressed before our DHT works, in particular churn (nodes joining and leaving without notice). To deal with churn the system needs some level of replication, i.e., a number of nodes which are responsible for each prefix such that failure of some nodes will not compromise the system. We give some more details in Section

14.4. In addition there are other issues (e.g., security, efficiency) which can be addressed to improve the system. Delay efficiency for instance is already considered in the seminal paper by Plaxton et al. These issues are beyond the scope of this lecture.

- The hypercube has many derivatives, the so-called *hypercubic networks*. Among these are the butterfly, cube-connected-cycles, shuffle-exchange, and de Bruijn graph. We start with the butterfly, which is basically a "rolled out" hypercube (hence directly providing replication!).

Definition 14.2 (Butterfly)**.** *Let $d \in N$. The* d-dimensional butterfly $BF(d)$ *is a graph with node set* $V = [d+1] \times [2]^d$ *and an edge set* $E = E_1 \cup E_2$ *with*

$$E_1 = \{\{(i,\alpha),(i+1,\alpha)\} \mid i \in [d],\ \alpha \in [2]^d\}$$

and

$$E_2 = \{\{(i,\alpha),(i+1,\beta)\} \mid i \in [d],\ \alpha,\beta \in [2]^d,\ \alpha \text{ and } \beta \text{ differ only at the } i^{th} \text{ position}\}\ .$$

A node set $\{(i,\alpha) \mid \alpha \in [2]^d\}$ *is said to form* level i *of the butterfly. The* d-dimensional wrap-around butterfly *W-BF(d) is defined by taking the* $BF(d)$ *and identifying level* d *with level 0.*

Remarks:

- Figure 14.3 shows the 3-dimensional butterfly $BF(3)$. The $BF(d)$ has $(d+1)2^d$ nodes, $2d \cdot 2^d$ edges and degree 4. It is not difficult to check that combining the node sets $\{(i,\alpha) \mid i \in [d]\}$ into a single node results in the hypercube.
- Butterflies have the advantage of a constant node degree over hypercubes, whereas hypercubes feature more fault-tolerant routing.
- The structure of a butterfly might remind you of sorting networks from Chapter 4. Although butterflies are used in the P2P context (e.g. Viceroy), they have been used decades earlier for communication switches. The well-known Benes network is nothing but two back-to-back butterflies. And indeed, butterflies (and other hypercubic networks) are even older than that; students familiar with fast fourier transform (FFT) will recognize the structure without doubt. Every year there is a new application for which a hypercubic network is the perfect solution!
- Indeed, hypercubic networks are related. Since all structured P2P architectures are based on hypercubic networks, they in turn are all related.
- Next we define the cube-connected-cycles network. It only has a degree of 3 and it results from the hypercube by replacing the corners by cycles.

Definition 14.3 (Cube-Connected-Cycles)**.** *Let $d \in N$. The* cube-connected-cycles *network CCC(d) is a graph with node set* $V = \{(a,p) \mid a \in [2]^d, p \in [d]\}$ *and edge set*

$$E = \{\{(a,p),(a,(p+1) \bmod d)\} \mid a \in [2]^d, p \in [d]\} \cup \{\{(a,p),(b,p)\} \mid a,b \in [2]^d, p \in [d], a = b \text{ except for } a_p\}\ .$$

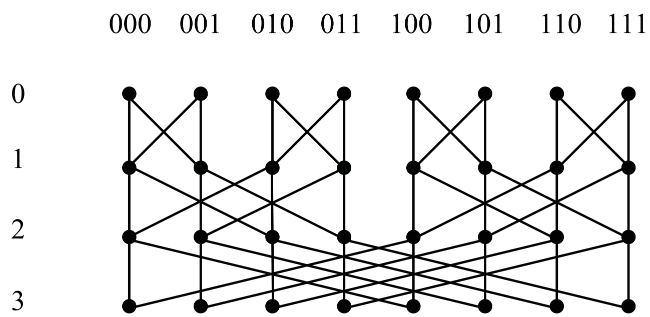

Figure 14.3: The structure of BF(3).

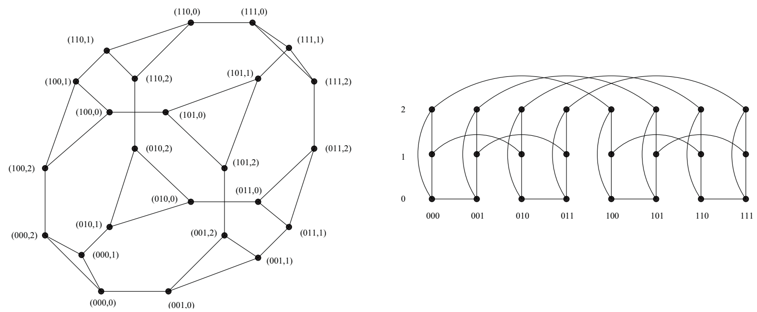

Figure 14.4: The structure of CCC(3).

Remarks:

- Two possible representations of a CCC can be found in Figure 14.4.
- The shuffle-exchange is yet another way of transforming the hypercubic interconnection structure into a constant degree network.

Definition 14.4 (Shuffle-Exchange)**.** *Let $d \in N$. The* d-dimensional shuffle-exchange $SE(d)$ *is defined as an undirected graph with node set* $V = [2]^d$ *and an edge set* $E = E_1 \cup E_2$ *with*

$$E_1 = \{\{(a_1,\ldots,a_d),(a_1,\ldots,\bar{a}_d)\} \mid (a_1,\ldots,a_d) \in [2]^d,\ \bar{a}_d = 1 - a_d\}$$

and

$$E_2 = \{\{(a_1,\ldots,a_d),(a_d,a_1,\ldots,a_{d-1})\} \mid (a_1,\ldots,a_d) \in [2]^d\}\ .$$

Figure 14.5 shows the 3- and 4-dimensional shuffle-exchange graph.

Definition 14.5 (DeBruijn)**.** *The* b-ary DeBruijn graph of dimension d *DB(b, d) is an undirected graph* $G = (V,E)$ *with node set* $V = \{v \in [b]^d\}$

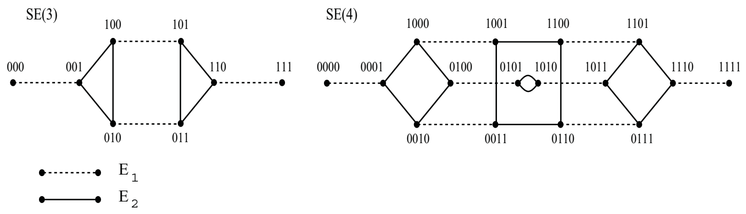

Figure 14.5: The structure of SE(3) and SE(4).

and edge set E that contains all edges $\{v, w\}$ with the property that $w \in \{(x, v_1, \ldots, v_{d-1}) : x \in [b]\}$, where $v = (v_1, \ldots, v_d)$.

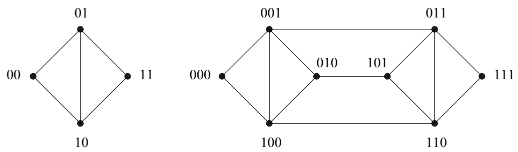

Figure 14.6: The structure of $DB(2, 2)$ and $DB(2, 3)$.

Remarks:

- Two examples of a DeBruijn graph can be found in Figure 14.6. The DeBruijn graph is the basis of the Koorde P2P architecture.
- There are some data structures which also qualify as hypercubic networks. An obvious example is the Chord P2P architecture, which uses a slightly different hypercubic topology. A less obvious (and therefore good) example is the skip list, the balanced binary search tree for the lazy programmer:

Definition 14.6 (Skip List)**.** *The skip list is an ordinary ordered linked list of objects, augmented with additional forward links. The ordinary linked list is the level 0 of the skip list. In addition, every object is promoted to level 1 with probability 1/2. As for level 0, all level 1 objects are connected by a linked list. In general, every object on level i is promoted to the next level with probability 1/2. A special start-object points to the smallest/first object on each level.*

Remarks:

- Search, insert, and delete can be implemented in $O(\log n)$ expected time in a skip list, simply by jumping from higher levels to lower ones when overshooting the searched position. Also, the amortized memory cost of each object is constant, as on average an object only has two forward pointers.

- The randomization can easily be discarded, by deterministically promoting a constant fraction of objects of level i to level $i + 1$, for all i. When inserting or deleting, object o simply checks whether its left and right level i neighbors are being promoted to level $i + 1$. If none of them is, promote object o itself. Essentially we establish a MIS on each level, hence at least every third and at most every second object is promoted.
- There are obvious variants of the skip list, e.g., the skip graph. Instead of promoting only half of the nodes to the next level, we always promote all the nodes, similarly to a balanced binary tree: All nodes are part of the root level of the binary tree. Half the nodes are promoted left, and half the nodes are promoted right, on each level. Hence on level i we have have 2^i lists (or, more symmetrically: rings) of about $n/2^i$ objects. This is pretty much what we need for a nice hypercubic P2P architecture.
- One important goal in choosing a topology for a network is that it has a small diameter. The following theorem presents a lower bound for this.

Theorem 14.7. *Every graph of maximum degree $d > 2$ and size n must have a diameter of at least $\lceil(\log n)/(\log(d - 1))\rceil - 2$.*

Proof. Suppose we have a graph $G = (V, E)$ of maximum degree d and size n. Start from any node $v \in V$. In a first step at most d other nodes can be reached. In two steps at most $d \cdot (d-1)$ additional nodes can be reached. Thus, in general, in at most k steps at most

$$1 + \sum_{i=0}^{k-1} d \cdot (d-1)^i = 1 + d \cdot \frac{(d-1)^k - 1}{(d-1) - 1} \le \frac{d \cdot (d-1)^k}{d-2}$$

nodes (including v) can be reached. This has to be at least n to ensure that v can reach all other nodes in V within k steps. Hence,

$$(d-1)^k \ge \frac{(d-2) \cdot n}{d} \quad \Leftrightarrow \quad k \ge \log_{d-1}((d-2) \cdot n/d) .$$

Since $\log_{d-1}((d-2)/d) > -2$ for all $d > 2$, this is true only if $k \ge \lceil(\log n)/(\log(d-1))\rceil - 2$. $\square$

Remarks:

- In other words, constant-degree hypercubic networks feature an asymptotically optimal diameter.
- There are a few other interesting graph classes, e.g., expander graphs (an expander graph is a sparse graph which has high connectivity properties, that is, from every not too large subset of nodes you are connected to a larger set of nodes), or small-world graphs (popular representations of social networks). At first sight hypercubic networks seem to be related to expanders and small-world graphs, but they are not.

14.4 DHT & Churn

As written earlier, a DHT essentially is a hypercubic structure with nodes having identifiers such that they span the ID space of the objects to be stored. We described the straightforward way how the ID space is mapped onto the peers for the hypercube. Other hypercubic structures may be more complicated: The butterfly network, for instance, may directly use the $d+1$ layers for replication, i.e., all the $d+1$ nodes with the same ID are responsible for the same hash prefix. For other hypercubic networks, e.g., the pancake graph (see exercises), assigning the object space to peer nodes may be more difficult.

In general a DHT has to withstand churn. Usually, peers are under control of individual users who turn their machines on or off at any time. Such peers join and leave the P2P system at high rates ("churn"), a problem that is not existent in orthodox distributed systems, hence P2P systems fundamentally differ from old-school distributed systems where it is assumed that the nodes in the system are relatively stable. In traditional distributed systems a single unavailable node is a minor disaster: all the other nodes have to get a consistent view of the system again, essentially they have to reach consensus which nodes are available. In a P2P system there is usually so much churn that it is impossible to have a consistent view at any time.

Most P2P systems in the literature are analyzed against an adversary that can crash a fraction of random peers. After crashing a few peers the system is given sufficient time to recover again. However, this seems unrealistic. The scheme sketched in this section significantly differs from this in two major aspects. First, we assume that joins and leaves occur in a worst-case manner. We think of an adversary that can remove and add a bounded number of peers; it can choose which peers to crash and how peers join. We assume that a joining peer knows a peer which already belongs to the system. Second, the adversary does not have to wait until the system is recovered before it crashes the next batch of peers. Instead, the adversary can constantly crash peers, while the system is trying to stay alive. Indeed, the system is *never fully repaired* but *always fully functional*. In particular, the system is resilient against an adversary that continuously attacks the "weakest part" of the system. The adversary could for example insert a crawler into the P2P system, learn the topology of the system, and then repeatedly crash selected peers, in an attempt to partition the P2P network. The system counters such an adversary by continuously moving the remaining or newly joining peers towards the sparse areas.

Clearly, we cannot allow the adversary to have unbounded capabilities. In particular, in any constant time interval, the adversary can at most add and/or remove $O(\log n)$ peers, n being the total number of peers currently in the system. This model covers an adversary which repeatedly takes down machines by a distributed denial of service attack, however only a logarithmic number of machines at each point in time. The algorithm relies on messages being delivered timely, in at most constant time between any pair of operational peers, i.e., the synchronous model. Using the trivial synchronizer this is not a problem. We only need bounded message delays in order to have a notion of time which is needed for the adversarial model. The duration of a round is then proportional to the propagation delay of the slowest message.

In the remainder of this section, we give a sketch of the system: For simplicity, the basic structure of the P2P system is a hypercube. Each peer is part of a distinct hypercube node; each hypercube node consists of $\Theta(\log n)$ peers. Peers have connections to other peers of their hypercube node and to peers of the neighboring hypercube nodes.[1] Because of churn, some of the peers have to change to another hypercube node such that up to constant factors, all hypercube nodes own the same number of peers at all times. If the total number of peers grows or shrinks above or below a certain threshold, the dimension of the hypercube is increased or decreased by one, respectively.

The balancing of peers among the hypercube nodes can be seen as a dynamic token distribution problem on the hypercube. Each node of the hypercube has a certain number of tokens, the goal is to distribute the tokens along the edges of the graph such that all nodes end up with the same or almost the same number of tokens. While tokens are moved around, an adversary constantly inserts and deletes tokens. See also Figure 14.7.

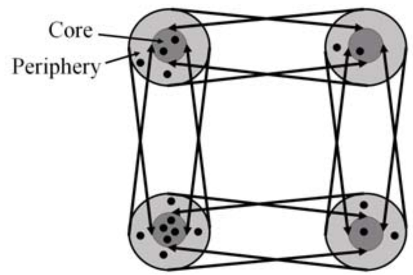

Figure 14.7: A simulated 2-dimensional hypercube with four nodes, each consisting of several peers. Also, all the peers are either in the core or in the periphery of a node. All peers within the same node are completely connected to each other, and additionally, all peers of a node are connected to the core peers of the neighboring nodes. Only the core peers store data items, while the peripheral peers move between the nodes to balance biased adversarial changes.

In summary, the P2P system builds on two basic components: i) an algorithm which performs the described dynamic token distribution and ii) an information aggregation algorithm which is used to estimate the number of peers in the system and to adapt the dimension of the hypercube accordingly:

Theorem 14.8 (DHT with Churn)**.** *We have a fully scalable, efficient P2P system which tolerates* $O(\log n)$ *worst-case joins and/or crashes per constant time interval. As in other P2P systems, peers have* $O(\log n)$ *neighbors, and the usual operations (e.g., search, insert) take time* $O(\log n)$*.*

Remarks:

- Indeed, handling churn is only a minimal requirement to make a P2P system work. Later studies proposed more elaborate architectures which can also handle other security issues, e.g., privacy or Byzantine attacks.

[1] Having a logarithmic number of hypercube neighbor nodes, each with a logarithmic number of peers, means that each peers has $\Theta(\log^2 n)$ neighbor peers. However, with some additional bells and whistles one can achieve $\Theta(\log n)$ neighbor peers.

- It is surprising that unstructured (in fact, hybrid) P2P systems dominate structured P2P systems in the real world. One would think that structured P2P systems have advantages, in particular their efficient logarithmic data lookup. On the other hand, unstructured P2P networks are simpler, in particular in light of non-exact queries.

14.5 Storage and Multicast

As seen in the previous section, practical implementations often incorporate some non-rigid (flexible) part. In a system called Pastry, prefix-based overlay structures similar to hypercubes are used to implement a DHT. Peers maintain connections to other peers in the overlay according to the lengths of the shared prefixes of their respective identifiers, where each peer carries a d-bit peer identifier. Let β denote the number of bits that can be fixed at a peer to route any message to an arbitrary destination. For $i = \{0, \beta, 2\beta, 3\beta, \ldots\}$, a peer chooses, if possible, $2^\beta - 1$ neighbors whose identifiers are equal in the i most significant bits and differ in the subsequent β bits by one of $2^\beta - 1$ possibilities. If peer identifiers are chosen uniformly at random, the length of the longest shared prefix is bounded by $O(\log n)$ in an overlay containing n peers; thus, only $O(\log n(2^\beta - 1)/\beta)$ connections need to be maintained. Moreover, every peer reaches every other peer in $O(\frac{\log n}{\beta})$ hops by repetitively selecting the next hop to fix β more bits toward the destination peer identifier, yielding a logarithmic overlay diameter.

The advantage of prefix-based over more rigid DHT structures is that there is a large choice of neighbors for most prefixes. Peers are no longer bound to connect to peers exactly matching a given identifier. Instead peers are enabled to connect to any peer matching a desired prefix, regardless of subsequent identifier bits. In particular, among half of all peers can be chosen for a shared prefix of length 0. The flexibility of such a neighbor policy allows the optimization of secondary criteria. Peers may favor peers with a low-latency and select multiple neighbors for the same prefix to gain resilience against churn. Regardless of the choice of neighbors, the overlay always remains connected with a bounded degree and diameter.

Such overlay structures are not limited to distributed storage. Instead, they are equally well suited for the distribution of content, such as multicasting of radio stations or television channels. In a basic multicasting scheme, a source with identifier 00...0 may forward new data blocks to two peers having identifiers starting with 0 and 1. They in turn forward the content to peers having identifiers starting with 00, 01, 10, and 11. The recursion finishes once all peers are reached. This basic scheme has the subtle shortcoming that data blocks may pass by multiple times at a single peer because a predecessor can match a prefix further down in its distribution branch.

The subsequent multicasting scheme $\mathcal{M}$ avoids this problem by modifying the topology and using a different routing scheme. For simplicity, the neighbor selection policy is presented for the case $\beta = 1$. In order to use $\mathcal{M}$, the peers must store links to a different set of neighbors. A peer v with the identifier $b_0^v \ldots b_{d-1}^v$ stores links to peers whose identifiers start with $b_0^v b_1^v \ldots b_{i-1}^v \overline{b_i^v} b_{i+1}^v$ and $b_0^v b_1^v \ldots b_{i-1}^v \overline{b_i^v b_{i+1}^v}$ for all $i \in \{0, \ldots, d-2\}$. For example, the peer with the identifier 0000 has to maintain connections to peers whose identifiers start with the prefixes 10, 11, 010, 011, 0010, and 0011. Pseudo-code for the algorithm is given in Algorithm 49.

Algorithm 49 $\mathcal{M}$: forward(π, v_c) at peer v.

```
S := {v' ∈ N_v | ℓ(v', v) ≥ π + 1}
choose v_1 ∈ S: ℓ(v_1, v) ≤ ℓ(ṽ, v) ∀ṽ ∈ S
if v_1 ≠ ∅ then forward(ℓ(v_1, v), v) to v_1 fi
if v_c ≠ ∅ then
   choose v_2 ∈ N_v: ℓ(v_2, v_c) = π + 1
   if v_2 = ∅ then v_2 := getNext(v) from v_c fi
   if v_2 ≠ ∅ then forward(ℓ(v_2, v_c), v_c) to v_2 fi
else
   choose v_2 ∈ N_v: ℓ(v_2, v) = π
   if v_2 ≠ ∅ then forward(π + 1, v_c) to v_2 fi
fi
```

The parameters are the length π of the prefix that is not to be modified and at most one critical predecessor v_c. If $\beta = 1$, any node v tries to forward the data block to two peers v_1 and v_2. The procedure is called at the source v_0 with arguments $\pi := 0$ and $v_c := \emptyset$, resulting in the two messages *forward*$(1, v_0)$ to v_1 and *forward*$(1, \emptyset)$ to v_2. The peer v_1 is chosen locally such that the prefix its identifier shares with the identifier of v is the shortest among all those whose shared prefix length is at least $\pi + 1$. This value $\ell(v_1, v)$ and v itself are the parameters included in the forward message to peer v_1, if such a peer exists. The second peer is chosen similarly, but with respect to v_c and not v itself. If no suitable peer is found in the routing table, the peer v_c is queried for a candidate using the subroutine *getNext* which is described in Algorithm 50. This step is required because node v cannot deduce from its routing table whether a peer v_2 with the property $\ell(v_2, v_c) \geq \pi + 1$ exists. In the special case when $v_c = \emptyset$, v_2 is chosen locally, if possible, such that $\ell(v_2, v) = \pi$. In Figure 14.8, a sample spanning tree resulting from the execution of $\mathcal{M}$ is depicted.

Algorithm 50 getNext(v_s) at peer v

```
S := {v' ∈ N_v | ℓ(v', v) > ℓ(v_s, v)}
choose v_r ∈ S: ℓ(v_r, v) ≤ ℓ(ṽ, v) ∀ṽ ∈ S
send v_r to v_s
```

The presented multicasting scheme $\mathcal{M}$ has the property that, at least in a static setting, wherein peers neither join nor leave the overlay, all peers can be reached and each peer receives a data block exactly once as summarized by the following theorem:

Theorem 14.9. *In a static overlay, algorithm $\mathcal{M}$ has the following properties:*

(a) It does not induce any duplicate messages (loop-free), and

(b) all peers are reached (complete).

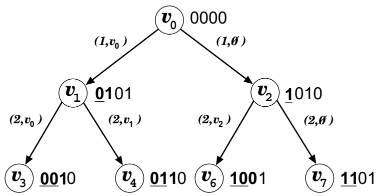

Figure 14.8: The spanning tree induced by a forward message initiated at peer v_0 is shown. The fixed prefix is underlined at each peer, whereas prefixes in bold print indicate that the parent peer has been constrained to forward the packet to peers with these prefixes.

Remarks:

- The multicast scheme $\mathcal{M}$ benefits from the same overlay properties as DHTs; there is a bounded diameter and peer degree. Peers can maintain backup neighbors and favor low-latency, high-bandwidth peers as neighbors. Most importantly, intermediate peers have the possibility to choose among multiple (backup) neighbors to forward incoming data blocks. This, in turn, allows peers to quickly adapt to changing network conditions such as churn and congestion. It is not necessary to rebuild the overlay structure after failures. In doing so, a system can gain both robustness and effiency.

- In contrast, for more rigid data structures, such as trees, data blocks are forced to travel along fixed data paths, rendering them susceptible to any kind of failure.

- Conversely, unstructured and more random overlay networks lack the structure to immediately forward incoming data blocks. Instead, such systems have to rely on the exchange of periodic notifications about available data blocks and requests and responses for the download of missing blocks, significantly increasing distribution delays. Furthermore, the lack of structure makes it hard to maintain connectivity among all peers. If the neighbor selection is not truly random, but based on other criertia such as latency and bandwidth, clusters may form that disconnect themselves from the remaining overlay.

There is a varierty of further flavors and optimizations for prefix-based overlay structures. For example, peers have a logarithmic number of neighbors in the presented structure. For $100,000$ and more peers, peers have at least 20 neighbors. Selecting a backup neighbor doubles the number of neighbors to 40. Using $\mathcal{M}$ further doubles their number to 80. A large number of neighbors accrues substantial maintenance costs. The subsequent variation limits the number of neighbors with a slight adjustment of the overlay structure. It organizes peers into disjoint groups $\mathcal{G}_0, \mathcal{G}_1, \dots, \mathcal{G}_m$ of about equal size. The introduction of groups is motivated by the fact that they will enable peers to have neighboring connections for a subset of all shared prefixes while maintaining the favorable overlay properties. The techniques in use are related to a proposal called LAND,

a locality-aware distributed hash table proposed by Abraham et al. The source, feeding blocks into the overlay, joins group $\mathcal{G}_0$. The other peers randomly join groups. Let $g(v)$ denote the function that assigns each peer v to a group, i.e., $v \in \mathcal{G}_{g(v)}$.

Peers select neighboring peers based not solely on shared prefixes but also on group membership. A peer v with the identifier $b_0^v \dots b_{d-1}^v$ stores links to neighboring peers whose identifiers start with $b_0^v b_1^v \dots b_{i-1}^v \overline{b_i^v}$ and belong to group $g(v) + 1 \bmod m$ for all $i \in \{g(v), g(v)+m, g(v)+2m, g(v)+3m, \dots\}$. Furthermore, let f denote the first index i where no such peer exists. As fallback, peer v stores further links to peers from arbitrary groups whose identifiers start with $b_0^v b_1^v \dots b_{k-1}^v \overline{b_k^v}$ for all $k \geq f - m + 1$. The fallback connections allow a peer to revert to the regular overlay structure for the longest shared prefixes where only few peers exist.

As an example, a scenario with $m = 4$ groups is considered. A peer with identifier $00\dots0$ belonging to group $\mathcal{G}_2$ has to maintain connections to peers from group $\mathcal{G}_3$ that share the prefixes 001, 0000001, 00000000001, etc. In an overlay with 100 peers, the peer is unlikely to find a neighbor for a prefix length larger than log(100), such as prefix 00000000001. Instead, he further maintains fallback connections to peers from arbitrary groups having identifiers starting with the prefixes 00000001, 000000001, 000000001, etc. (if such peers exist).

Remarks:

- By applying the presented grouping mechanism, the total number of neighbors is reduced to $\frac{2 \log n}{m} + c$ with constant c for fallback connections. (Note that peers have both outgoing neighbors to the next group and incoming neighbors from the previous group, doubling the number of neighbors.)

- Setting the number of groups $m = \log n$ gives a constant number of neighbors regardless of the overlay size.