Reliability of Distributed Systems*
Lecture 12: Introduction to Peer-to-Peer systems

Notes taken by Shay Horovitz

January 13, 2005

Summary: In this lesson, we'll learn about the basics of Peer-to-Peer (P2P) computing. We'll start by a general introduction to the field, continue with a scan of the most popular techniques being used today and emphasize systems that are based on a Distributed Hash Table (DHT). Then we'll summarize some basic routing schemes and slightly touch the topic of dynamic graph building.

1 Introduction to Peer-to-Peer

In traditional computer systems, all data was hosted and handled by a single server or server farms. This approach was also termed as Client/Server infrastructure. Throughout the design process of those systems, computer designers used to devote the main efforts in the scalability and the robustness of the server side. The Client/Server approach was and still is very common in computer systems; However, the price of maintaining the server scalability becomes very expensive and even not realistic as the number of clients increases tremendously.

A new approach called Peer-to-Peer (P2P) offers a distributed solution where there is no central server that hosts all data. Instead, all data is distributed among nodes (the former clients). In typical P2P systems, there will be no centralized control or database and no peer will have a global view of the whole system. In addition, there will be many nodes and among them some will be unreliable. P2P nodes have the ability to self organize the data between them, while maintaining fault tolerance.

Another interesting definition of P2P systems is by the relativity of computers, as defined in Webopedia: "P2P is a type of network in which each workstation has equivalent capabilities and responsibilities. This differs from Client/Server architectures, in which some computers are dedicated to serving the others".

While P2P became popular with the entrance of Napster to the market, it is not new at all, at least technically. For example, routers are a good example of a P2P system, as they discover the network topology and maintain it, fault tolerant and autonomous, with no server ruling. Other close examples are the Usenet news systems and the DNS.

* Spring 2004, the Hebrew University, Israel.

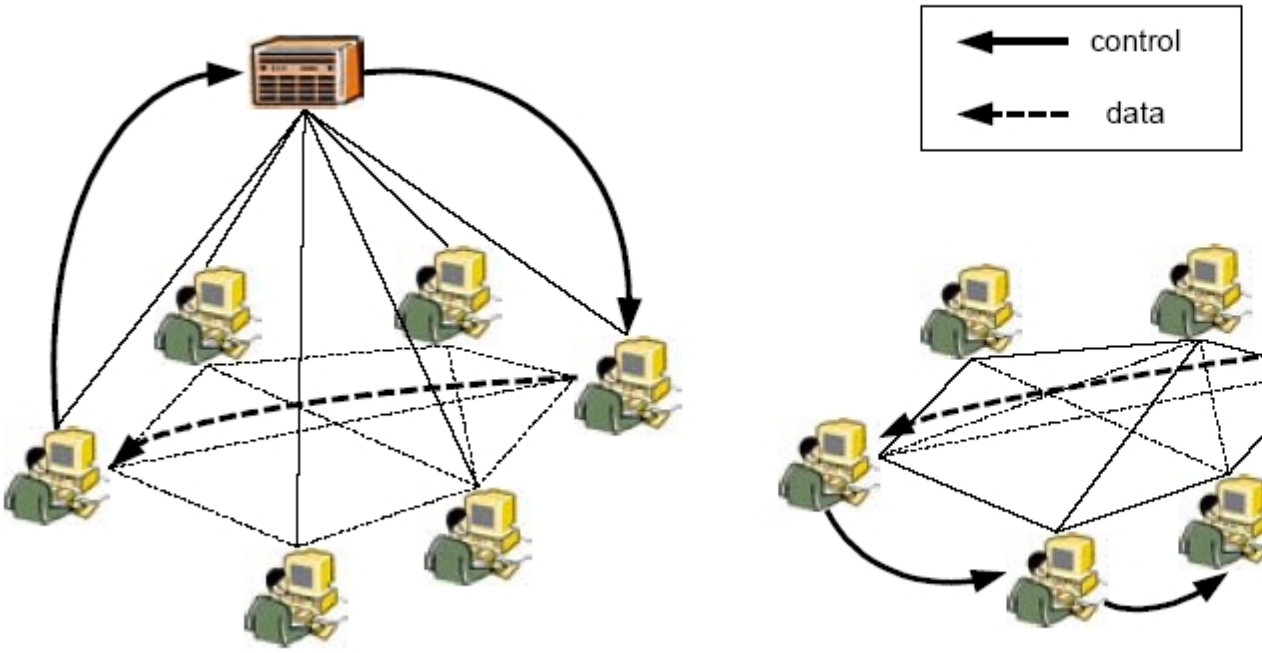


Figure 1: Three Peer-to-Peer architectures: Centralized(Left), Decentralized(Middle) and Hierarchical(Right)

Primary ”modern” designs of P2P systems in the late 90’s such as the file sharing services Napster and iMesh, used a dedicated server/ server farm as an index for the distributed data. Each client/node approached the server with a file id (usually a file name or MD5 digest of the file) and the server supplied a list of IP addresses of the nodes that share the requested file. The main disadvantage of Napster and iMesh was that by closing a single server, the whole system is down. This approach is also known as the Centralized P2P architecture.

A decentralized P2P architecture appeared with Nullsoft’s Gnutella, where there was no server at all. In Gnutella, a node requests a file from a small group of computers that it knows currently; then those nodes deliver the request to the nodes they know and so on. A limiting rule allowed a predefined maximal number of hops for each request route. Yet, Gnutella suffered from slow searches and the existence of islands of sub-networks that weren’t connected to each other. In addition, it was found that 70% of

the nodes shared no files and 50% of the searches responded from 1% of the hosts.

The latest major architecture being used in P2P systems today was first implemented in Fasttrack's KaZaA and is referred as the hierarchical P2P architecture. The key innovation of KaZaA is the use of Supernodes. Each node is either a supernode or is assigned to a supernode. Nodes with more bandwidth and better availability are automatically designated as supernodes. Each supernode knows about many other supernodes. A supernode searches the requests within its list of nodes and if needed, turns to other supernodes with the request. In KaZaA each file can be downloaded from multiple nodes in parallel, as implemented earlier in iMesh. Another system that implemented a KaZaA-like hierarchical architecture is Gnutella2.

2 Distributed Hash Tables Basics

A primary challenge in designing a P2P system is the problem of locating content. A simple strategy that was implemented in is an expanding ring search until the content is found. The cost of this strategy is at least N/r - where N is the number of nodes and r is the number of nodes that have a copy of the requested content. Another issue is that we need many copied of the content to keep the overhead small. A different strategy was led by Napster and iMesh, where there exists a centralized index. However, both systems faced the problems of high load on the server side and the existence of a single point of failure. As for locating content, KaZaA's solution can be seen as a combination of Gnutella and Napster, where inside the group of nodes, the supernode acts like a Napster server and outside a Gnutella-like solution ruling.

In order to solve the typical problems of popular P2P solutions, a "distributed" approach might lead us to assign particular nodes to hold particular content. We'd like that when a node requests for a content, it will go to the node that is supposed to hold it, or at least a node that knows where this content is located. We'd like to reach this under a minimal amount of bottlenecks - that is: we'd like to distribute the responsibilities evenly among the existing nodes as we can. In addition, our solution should also be adapted to nodes joining and leaving or failing. This means that whenever a node joins our system network, we should give it some responsibilities, and as it leaves of fails, we should redistribute its responsibilities among existing nodes. We'll describe a solution based on Distributed Hash Tables and analyze its characteristics.

A distributed hash table (DHT), is a technology based on hash tables enabling identification and retrieving, in distributed systems like some P2P networks, of information. The whole table is distributed on the network: each node has a part of it. A Hash Table associates data with keys. A key is hashed to find a bucket within the hash table and each bucket is expected to hold $items/buckets$ items. In a Distributed Hash Table, nodes play the role of the hash buckets. A key is hashed to find a responsible node node, while the data and the load are balanced across nodes. While DHTs smell like a good solution for locating content, we still need some modifications to solve some minor problems as we'll notice forward.

One problem with implementing a simple hash function, is the problem of dynamicity - adding or removing nodes. If we'll use the trivial hash mod N function (where N is the number of nodes), virtually every key will change its location as the number

of nodes is being changed. In addition, we might not even know exactly the value of N. The solution here would be to define a fixed hash space, where all hash values fall within that space and do not depend on the number of nodes/peers/hash buckets. Each key goes to the node closest to its ID in the hash space. The idea behind DHT hashing is similar to consistent hashing, but in this case the IDs in the hash space are within the range of [0,1]. Another problem is that in typical hash tables, all nodes must be known to insert or lookup data. The solution here is for each node to know only a few neighbors, and the messages are routed through neighbors via multiple hops.

A good DHT design will make sure that for each object, the node/s responsible for that object should be reachable via a short path - meaning a small network diameter. In addition, the number of neighbors for each node should remain reasonable - leading to a small degree. The DHT routing mechanism should be decentralized where there's no single point of failure or bottlenecks. It should also gracefully handle nodes joining and leaving, and provide low stretch.

3 Distributed Hash Tables case studies

3.1 Chord

Chord (created by MIT), is designed to offer the functionality necessary to implement general-purpose systems while preserving flexibility. Chord is an efficient distributed lookup system based on consistent hashing. It provides a unique mapping between an identifier space and a set of nodes. A node can be a host or a process identified by an IP address and a port number; each node is associated with a Chord identifier a. Chord maps each identifier a to the node with the smallest identifier greater than a. This node is called the successor of a. By using an additional layer that translates high level names into Chord identifiers, Chord may be used as a powerful lookup service. Chord maps identifiers to successor nodes; the distributed hash table being used by Chord associates values (blocks) with identifiers, and the application provides a file system interface.

Technically, Chord is emulating a circular m-bit ID space for both keys and nodes. Each node ID is calculated by SHA-1 of its IP address. Each key ID is calculated by SHA-1 of the key. A key is mapped to the first node whose ID is equal or follows the key ID, meaning that each node is responsible for $O(K/N)$ keys and $O(K/N)$ keys move when a node joins or leaves. In Chord, each node knows only 2 other nodes on the ring: its successor and its predecessor. Lookup is achieved by forwarding requests around the ring through successor pointers, and it requires $O(N)$ hops.

As seen in Figure 5, assume that the system is in a stable state (all routing tables contain correct information) and a search is initiated at node 2 of Figure 5(a) for the successor of identifier 6. The largest node with an identifier smaller than 6 is 5. The target of the search, 6, is in the interval defined by 5 and its successor (7); therefore 7 is the returned value.

When a node joins the ring, the process is assembled of 3 major parts: first comes the initialization of all fingers of the new node, then the fingers of existing nodes are being updated and finally keys are being transferred from successor node to the new

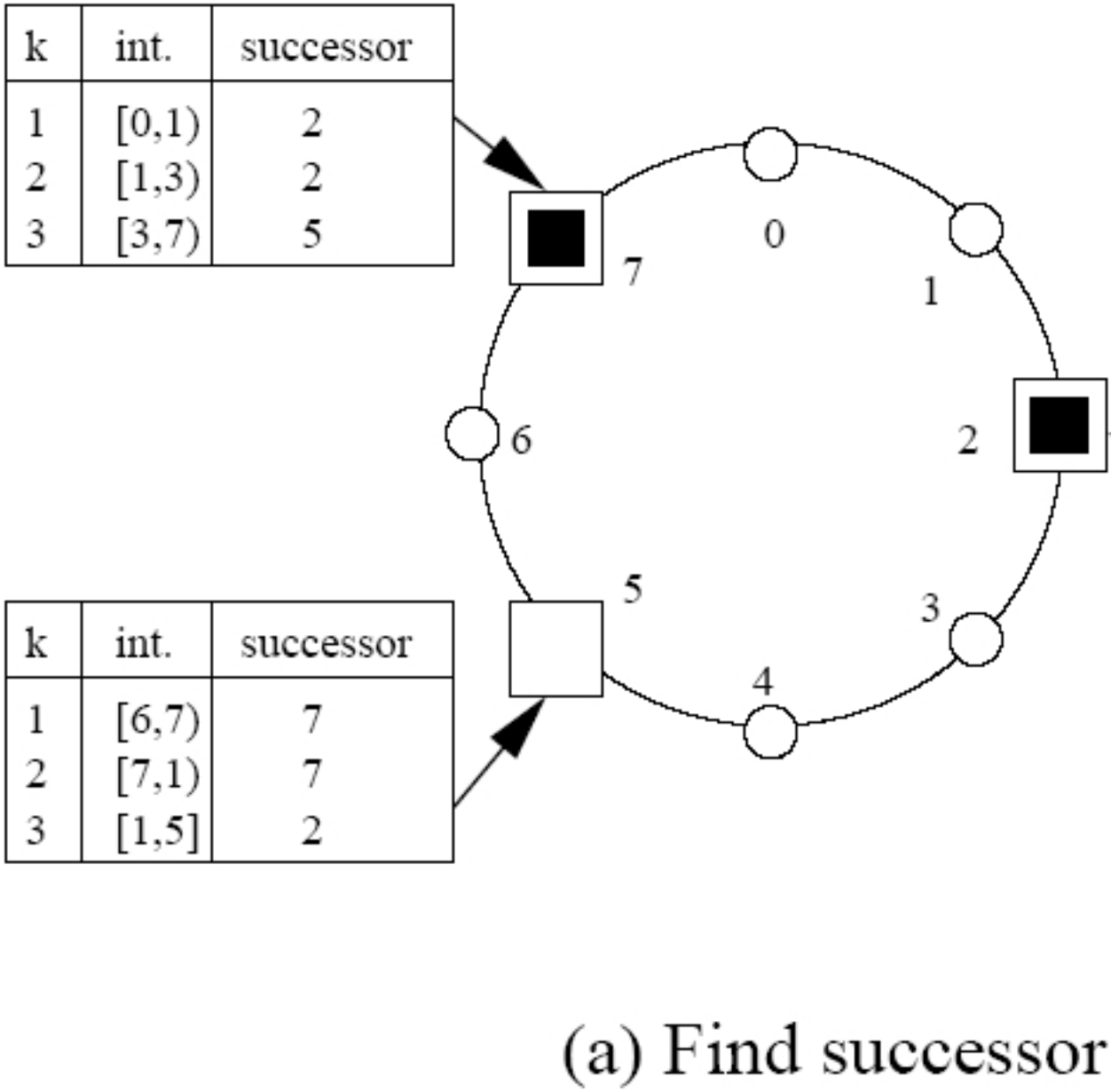


Figure 2: The Chord algorithm in a three-bit identifier space

node. A detailed description of Chord can be found in [DBK+01]

3.2 Pastry

Pastry (created by Microsoft Research) is a generic peer-to-peer content location and routing system based on a self-organizing overlay network of nodes connected via the Internet. Pastry is completely decentralized, fault-resilient, scalable, and reliably routes a message to the live node. pastry takes into account network locality; it seeks to minimize the distance messages travel, according to a to scalar proximity metric like the number of IP routing hops. However, pastry has a more complicated join protocol than Chord. A new node's routing table will be populated with information from nodes along the path taken by the join message - this leads to latency.

A detailed description of the Pastry protocol can be found in [RD01]; Yet we give a short review here. Each Pastry node has a unique, 128-bit nodeId. The set of existing nodeIds is uniformly distributed; this can be achieved, for instance, by basing the nodeId on a secure hash of the nodes public key or IP address. Given a message and a key, Pastry reliably routes the message to the Pastry node with the nodeId that is numerically closest to the key, among all live Pastry nodes. Assuming a Pastry network consisting of N nodes, Pastry can route to any node in less than $\lceil \log_{2^b} N \rceil$ steps on average (b is a configuration parameter with typical value 4). With concurrent node failures, eventual delivery is guaranteed unless $l/2$ or more nodes with *adjacent* nodeIds fail simultaneously (l is an even integer parameter with typical value 16).

The tables required in each Pastry node have only $(2^b - 1) * \lceil \log_{2^b} N \rceil + l$ entries, where each entry maps a nodeId to the associated nodes IP address. Moreover,after a node failure or the arrival of a new node, the invariants in all affected routing tables can be restored by exchanging $O(\log_{2^b} N)$ messages.

For the purposes of routing, nodeIds and keys are thought of as a sequence of digits with base 2^b. A nodes routing table is organized into $\lceil \log_{2^b} N \rceil$ rows with $2^b - 1$ entries each. The $2^b - 1$ entries in row n of the routing table each refer to a node whose nodeId matches the present nodes nodeId in the first n digits, but whose $n + 1$th digit has one of the 2bpossible values other than the n+1th digit in the present nodes id. The uniform distribution of nodeIds ensures an even population of the nodeId space; thus, only $\lceil \log_{2^b} N \rceil$ levels are populated in the routing table. Each entry in the routing table refers to one of potentially many nodes whose nodeId have the appropriate prefix. Among such nodes, the one closest to the present node (according to a scalar proximity metric, such as the round trip time) is chosen.

In addition to the routing table, each node maintains IP addresses for the nodes in its *leaf set*, i.e., the set of nodes with the $l/2$ numerically closest larger nodeIds, and the $l/2$ nodes with numerically closest smaller nodeIds, relative to the present nodes nodeId.

Figure 3 shows the path of an example message. In each routing step, the current node normally forwards the message to a node whose nodeId shares with the key a prefix that is at least one digit (or b bits) longer than the prefix that the key shares with the current nodeId. If no such node is found in the routing table, the message is forwarded to a node whose nodeId shares a prefix with the key as long as the current node, but is numerically closer to the key than the current nodeId. Such a node must exist in the

leaf set unless the nodeId of the current node or its immediate neighbour is numerically closest to the key, or l=2 adjacent nodes in the leaf set have failed concurrently.

A key design issue in Pastry is how to efficiently and dynamically maintain the node state, i.e., the routing table, leaf set and neighborhood sets, in the presence of node failures, node recoveries, and new node arrivals. Briefly, an arriving node with the newly chosen nodeId X can initialize its state by contacting a nearby node A (according to the proximity metric) and asking A to route a special message using X as the key. This message is routed to the existing node Z with nodeId numerically closest to X^1. X then obtains the leaf set from Z, and the ith row of the routing table from the ith node encountered along the route from A to Z. One can show that using this information, X can correctly initialize its state and notify nodes that need to know of its arrival.

To handle node failures, neighboring nodes in the nodeId space (which are aware of each other by virtue of being in each others leaf set) periodically exchange keepalive messages. If a node is unresponsive for a period T, it is presumed failed. All members of the failed nodes leaf set are then notified and they update their leaf sets. Since the leaf sets of nodes with adjacent nodeIds overlap, this update is trivial. A recovering node contacts the nodes in its last known leaf set, obtains their current leaf sets, updates its own leaf set and then notifies the members of its new leaf set of its presence. Routing table entries that refer to failed nodes are repaired lazily;

4 Basic Network Topologies and Routing

Let **O** be a group of objects. $h : \mathbf{O} \rightarrow [0..1]$ is a hash function (we assume that the mapping is done well). Each node *v* has an id: $v.id \in [0..1]$. We'll search for the object $o \in \mathbf{O}$ by finding a maximal id value that is lower than the value of the hash of o. That will be: $arg_{\forall v \in V} = \max \{ v.id | v.id < hash(o) \}$. First we'd like to build a network (network design) and then we'll construct a routing scheme.

4.1 Complete Graph

In a complete graph there's full connectivity - each node knows the id's of every other node in the graph, thus it takes 1 hop to reach any target. The disadvantage is that as the system gets larger, it should save many states and it also reflects for changes in a dynamic network.

4.2 Ring

In a ring, each node has exactly 2 neighbors. the disadvantage is the number of messages between the nodes. (we assume that the ring is sorted by IP addresses).

4.3 Binary Tree

In a binary tree, the degree is constant (3) and the diameter is $2 \lg n$. There's a problem of congestion. We check the probability that a node will participate in the routing of a

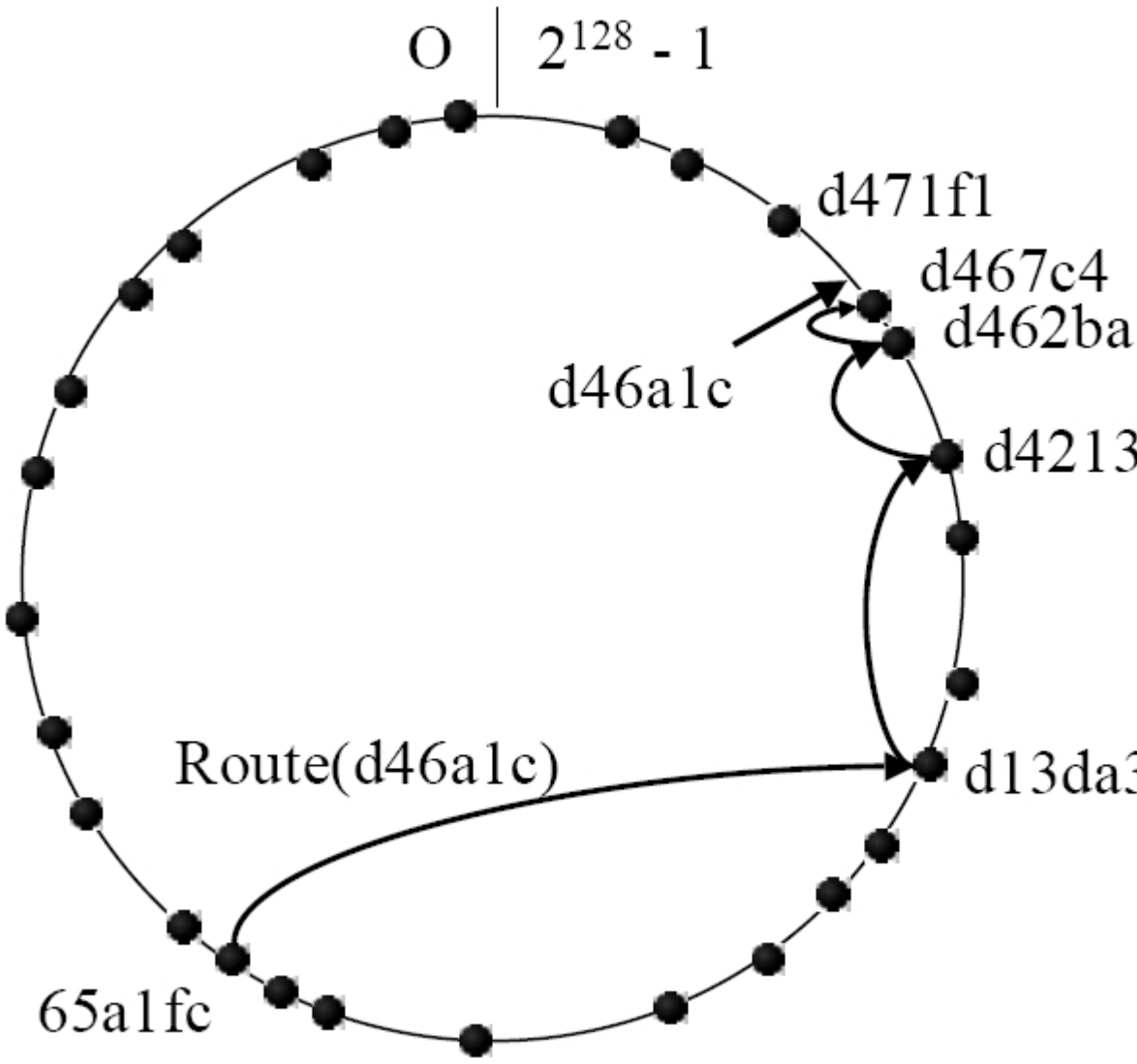
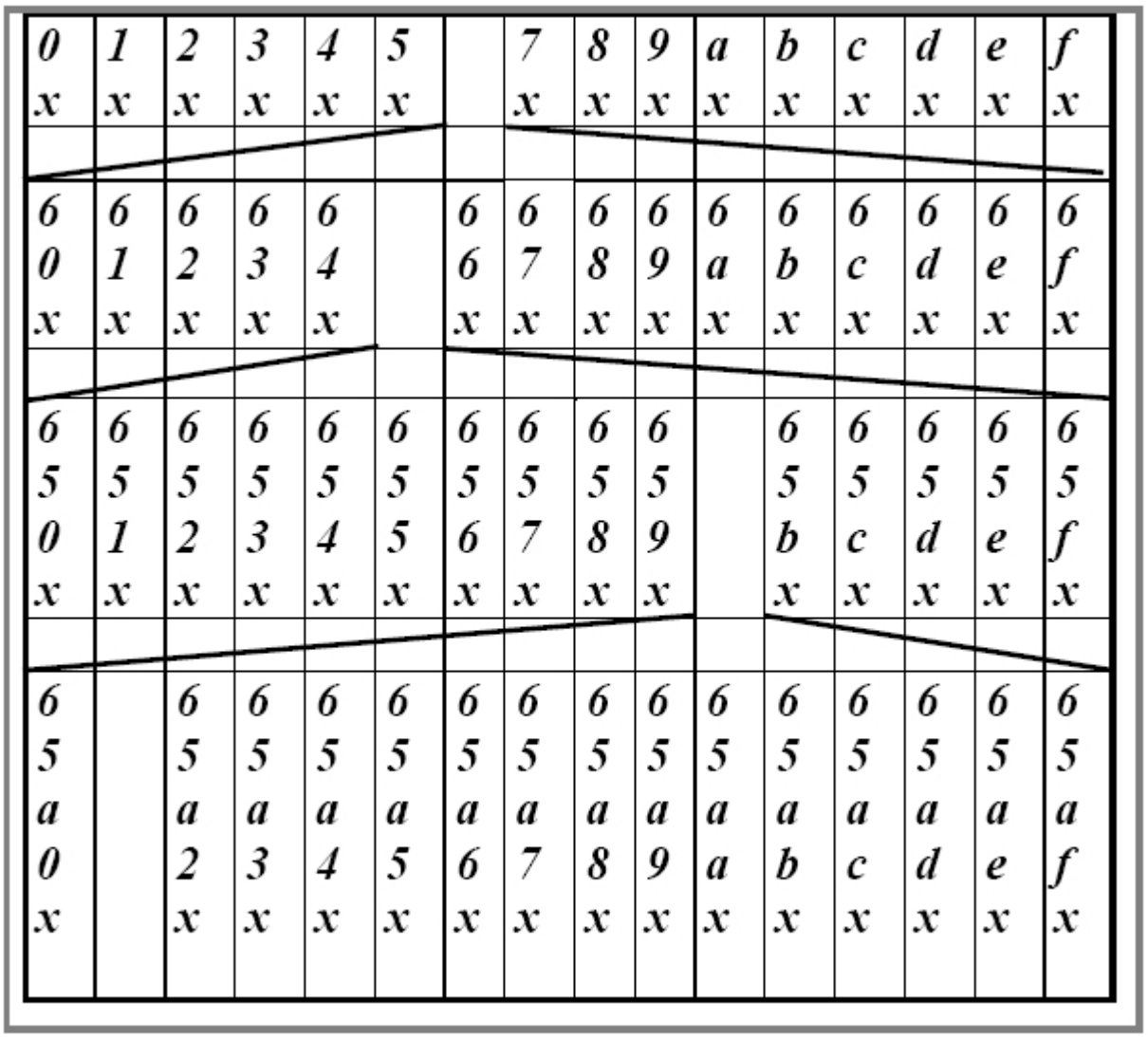


Figure 3: Routing a message from node 65a1fc with key d46a1c. The dots depict live nodes in Pastry's circular namespace



0	1	2	3	4	5		7	8	9	a	b	c	d	e	f
x	x	x	x	x	x		x	x	x	x	x	x	x	x	x
6	6	6	6	6		6	6	6	6	6	6	6	6	6	6
0	1	2	3	4		6	7	8	9	a	b	c	d	e	f
x	x	x	x	x		x	x	x	x	x	x	x	x	x	x
6	6	6	6	6	6	6	6	6	6		6	6	6	6	6
5	5	5	5	5	5	5	5	5	5		5	5	5	5	5
0	1	2	3	4	5	6	7	8	9		b	c	d	e	f
x	x	x	x	x	x	x	x	x	x		x	x	x	x	x
6		6	6	6	6	6	6	6	6	6	6	6	6	6	6
5		5	5	5	5	5	5	5	5	5	5	5	5	5	5
a		a	a	a	a	a	a	a	a	a	a	a	a	a	a
0		2	3	4	5	6	7	8	9	a	b	c	d	e	f
x		x	x	x	x	x	x	x	x	x	x	x	x	x	x

Figure 4: Routing table of a Pastry node with nodeId 65a1x,b=4. Digits are in base 16, x represents an arbitrary suffix. The IP address associated with each entry is not shown

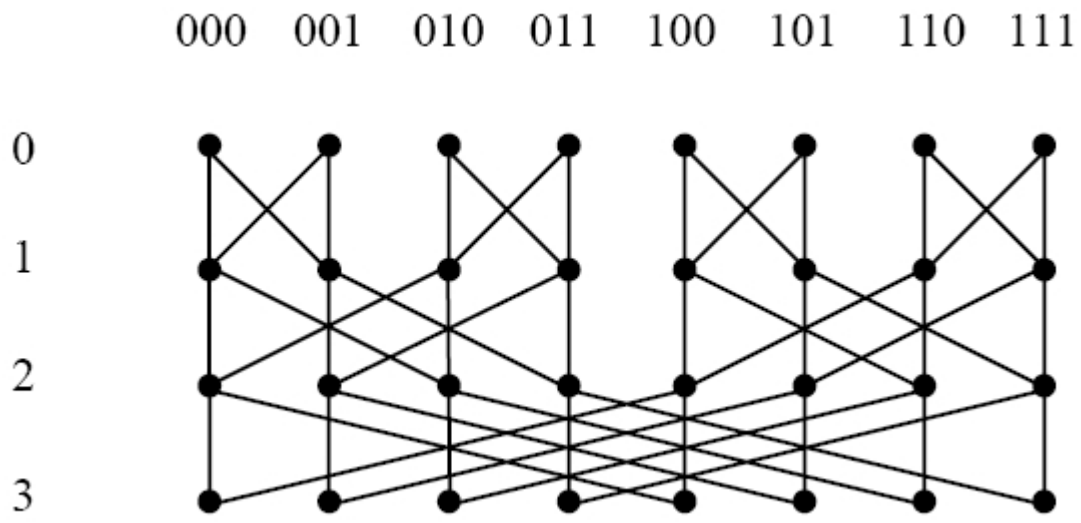


Figure 5: The structure of Butterfly BF(3)

random pair of source and destination nodes. With a probability of $\frac{1}{2}$ the node will be in the other side of the tree.

4.4 Butterfly

Let $d \in \mathbb{N}$. The d-dimensional butterfly $BF(d)$ is a graph with node set $V = [d+1] \times [2]^d$ and an edge set $E = E_1 \bigcup E_2$ with

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

and

$$E_2 = \{\{(i, \alpha), (i+1, \beta)\} | i \in [d], \alpha, \beta \in [2]^d\}$$

A node set $\{(i, \alpha) | \alpha \in [2]^d\}$ 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.

Figure 5 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 $(i, \alpha) | i \in [d]$ into a single node results in the hypercube.

4.5 Base b Hyper Cube

In a base-b hyper cube: $\mathbf{V} = \{a_1..a_n | a_i \in \{0..b-1\}\}$. In a binary hyper cube, each node is represented by 0's and 1's. For each $1 \leq i \leq k$ and for each $j \neq a_i$ exists an

arc $\{a_1..a_{i-1}, a_j, a_{i+1}, .., a_k\}$. The degree is $k(b-1)$. If the degree is $\lg_b k$ and the diameter is k, as $b = n^{\frac{1}{2}}$ then k is constant. Another solution is to fix a_i to $a_i \pm 1_{modb}$ - then the degree is 2k and the delimeter is $\frac{b}{2}$. For $b = \sqrt{n}$ we get a grid.

4.6 Base b Cube Connected Cycle

Let $d \in \mathbb{N}$. The cube-connected-cycles network $CCC(d)$ is a graph with node set $V = \{(a,p)|a \in [2]^d, p \in [d]\}$ and edge set

$$E = \{\{(a,p),(a,(p+1) \mod d)\}|a \in [2]^d, p \in [d]\}$$

$$\bigcup\{\{(a,p),(b,p)\}|a,b \in [2]^d, p \in [d], a = b \text{ except for } a_p\}$$

In a base-b cube connected cycle the number of nodes is $b^k k$

The degree is constant (3 in the binary case) with a diameter of 2.5k. $|V| = b^k k$ and $\lg n = A + \lg k$

4.7 Shuffle Exchange

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

$$E_1 = \{\{(a_{d-1}...a_0),(a_{d-1}...\overline{a}_0)\}|(a_{d-1}...a_0) \in [2]^d, \overline{a}_0 = 1 - a_0\}$$

and

$$E_2 = \{\{(a_{d-1}...a_0),(a_0 a_{d-1}...a_1)\}|(a_{d-1}...a_0) \in [2]^d\}$$

4.8 DeBruijn

The *b*-ary DeBruijn graph dimention *d DB(b,d)* is a undirected graph G=(V,E) with node set V=$\{v \in [b]^d\}$ and edge set *E* that contains all edges $\{v, w\}$ with the property that $w \in \{(x, v_{d-1}, ..., v_1) : x \in [b]\}$, where $v = (v_{d-1}, ..., v_0)$.

4.9 Building a Dynamic Graph

We'll set an id for each node, and the size of its id describes its "world". For example - a node with an id of size k thinks that he's inside a k-nodes cube. We'd like to connect the arcs between the nodes. Properties:

1. No node's id is a prefix of any other node's id
2. The ids of the nodes are being used as prefix codes

What happens when the tree is not complete ? - A node will connect to a node in a higher level (where a node is missing). To be more precise, we'll review the case of Dynamic Graphs as stated in [AAA$^+$03], which offers an algorithm for maintaining a dynamic overlay network that derives its characteristics from a family of static graphs.

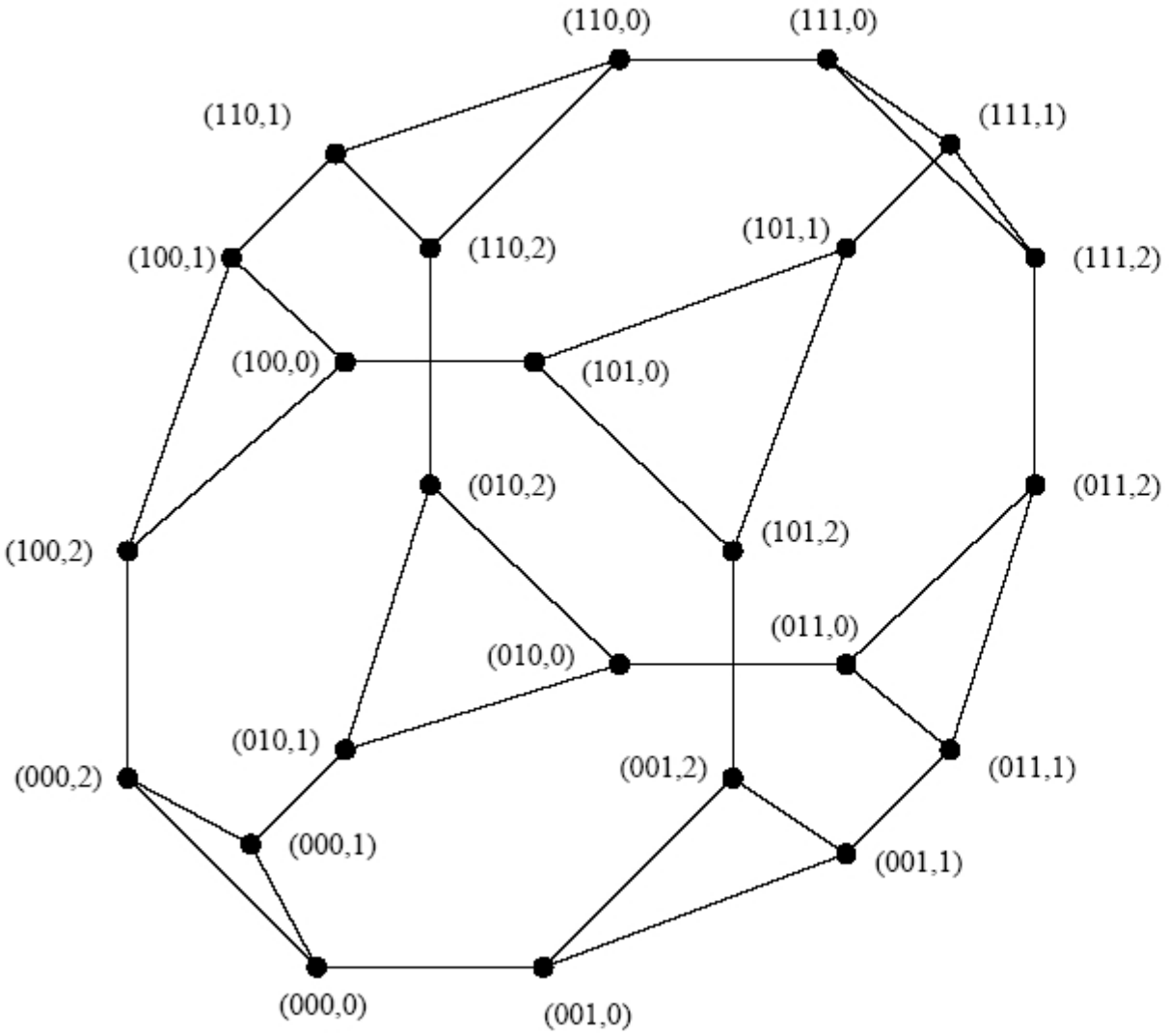


Figure 6: The structure of Cube Connected Cycles CCC(3)

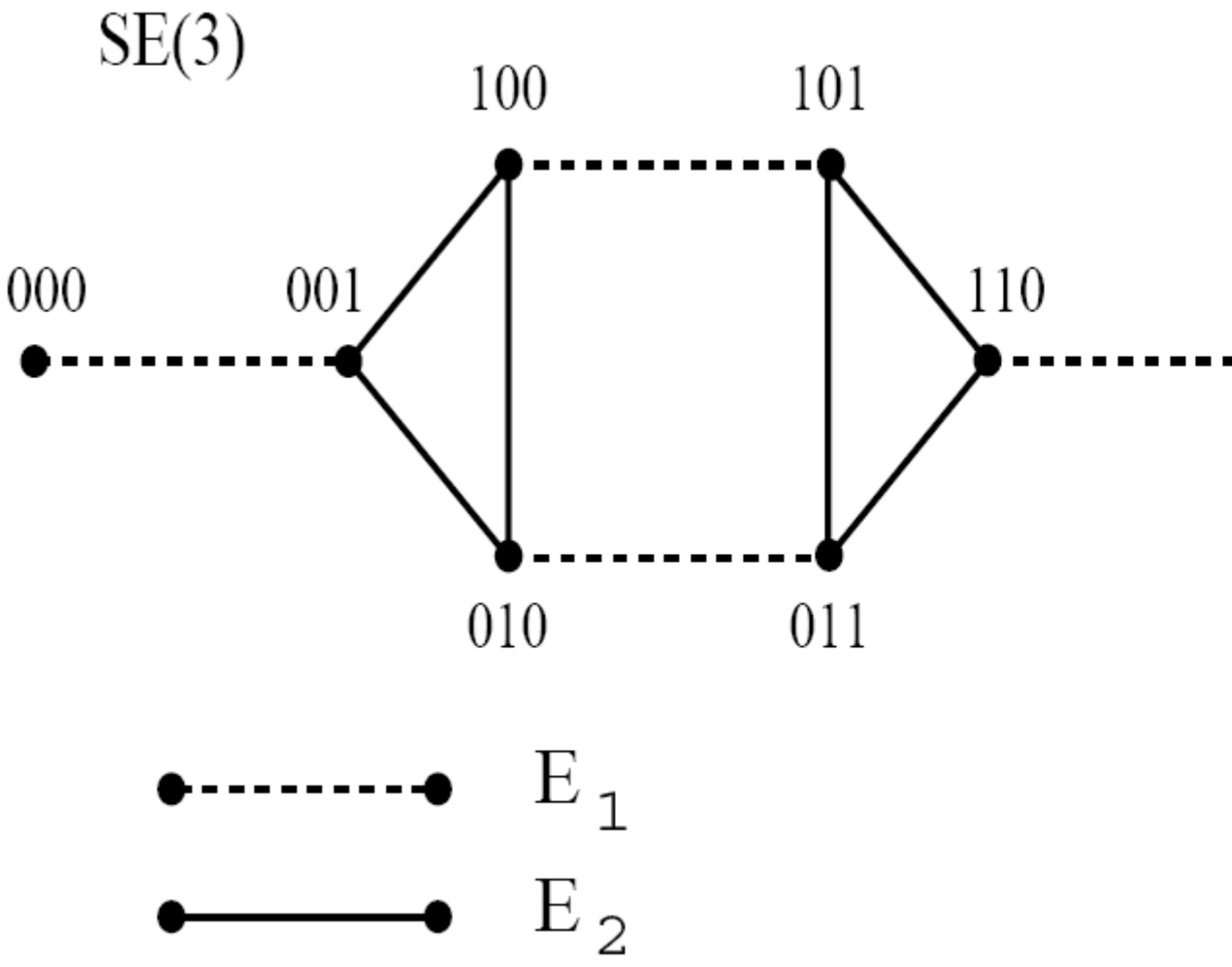
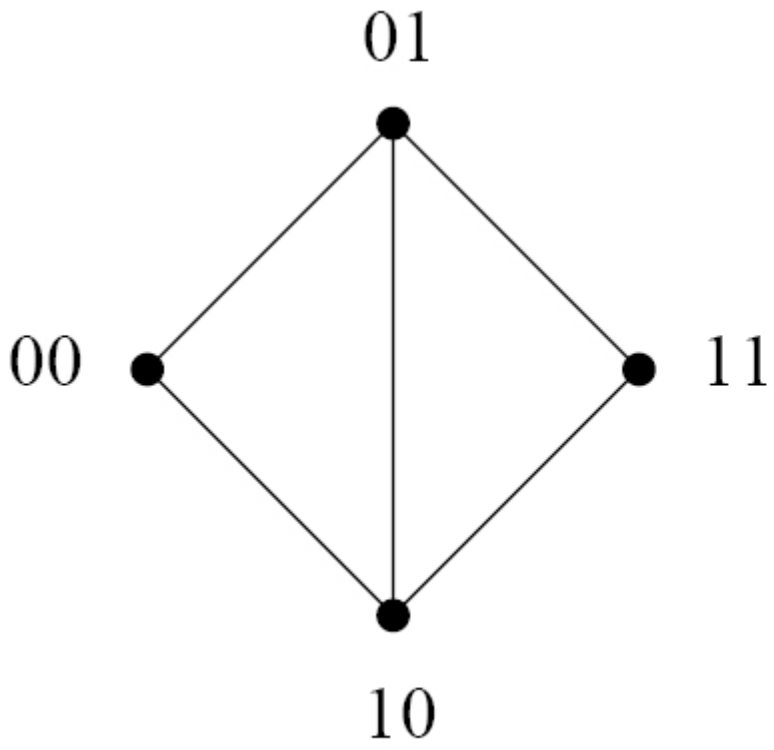


Figure 7: The structure of Shuffle Exchange SE(3) and SE(4)



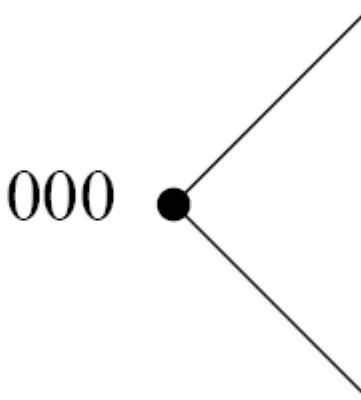


Figure 8: The structure of DeBruijn DB(2,2) and DB(2,3)

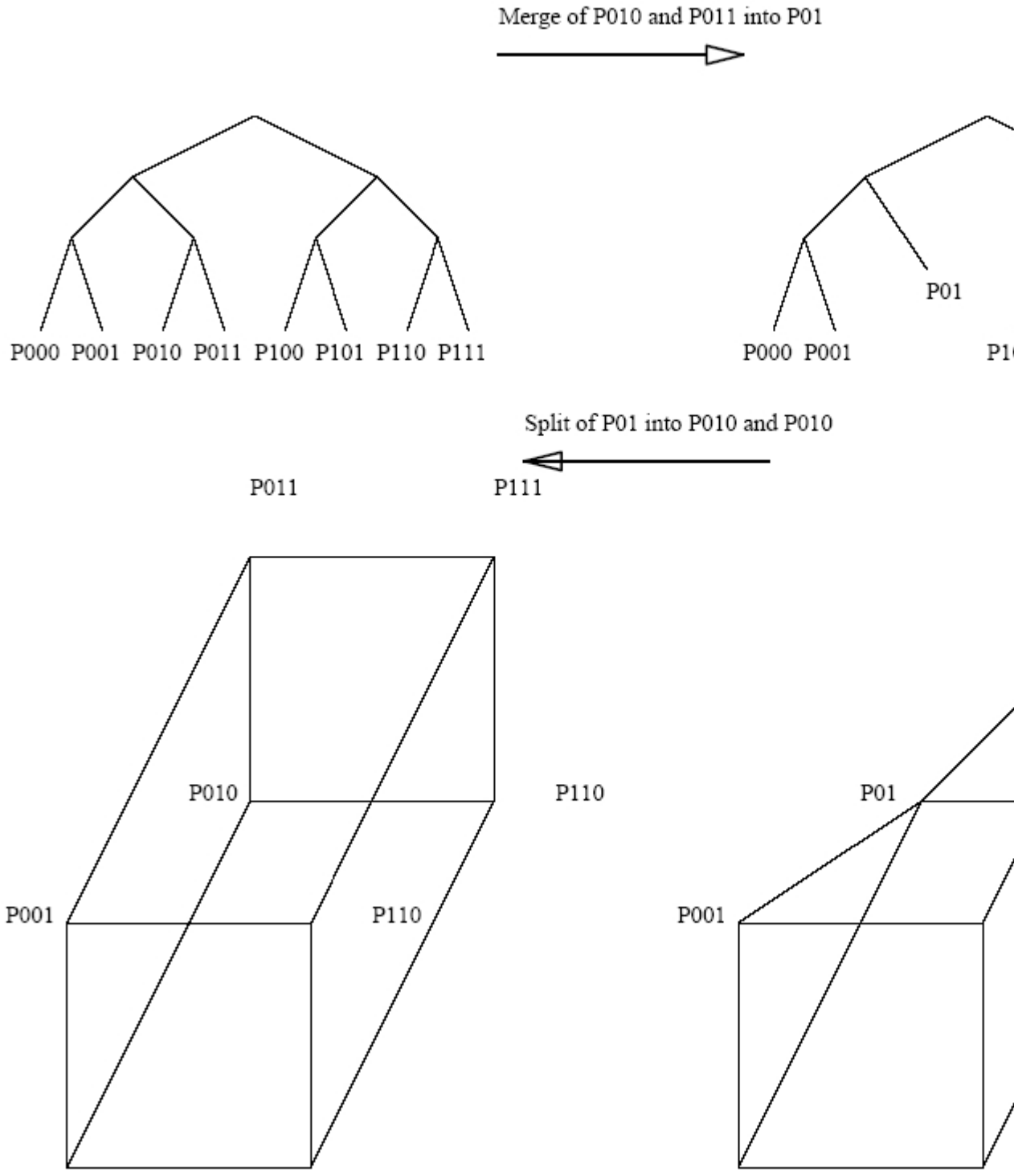


Figure 9: An example of merge and split on a dynamic hypercube: view of the dynamic graph as a tree (above) and the graph itself (bottom)

Our goal is to make use of family of graphs in order to maintain a dynamic graph that nodes can join and leave. Intuitively, this works by having each node join some location at G_i by splitting it into a set of children at G_{i+1}, and vice versa for leaving.

The nodes of the dynamic overlay graph can be thought of as the leaves of a tree. The inner vertexes represent nodes that no longer exist (were split), and the leaves represent current nodes. In order to maintain the tree, when a node joins the network, it chooses some location to join and "splits" it into leaves. On the other hand, when a node leaves the network, it finds a full set of siblings, and merges the remaining subset into a single parent. The algorithms are presented in the article.

In Figure 9 we show a merge and a split operation on a dynamic hypercube. It's easy to see that split and merge operations keep the dynamic graph properties.

In the article, you'll also find 3 techniques to balance the graph: one deterministic and one randomized, and the last is a combination of the first two techniques.

References

[AAA+03] Ittai Abraham, Baruch Awerbuch, Yossi Azar, Yair Bartal, Dahlia Malkhi, and Elan Pavlov. A generic scheme for building overlay networks in adversarial scenarios. In *Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 40.2. IEEE Computer Society, 2003.

[DBK+01] Frank Dabek, Emma Brunskill, M. Frans Kaashoek, David Karger, Robert Morris, Ion Stoica, and Hari Balakrishnan. Building peer-to-peer systems with chord, a distributed lookup service. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII)*, Schloss Elmau, Germany, May 2001. IEEE Computer Society.

[RD01] Antony Rowstron and Peter Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *Lecture Notes in Computer Science*, 2218:329–??, 2001.