UNIVERSITÀ DEGLI STUDI DI PADOVA

Facoltà di Ingegneria
Corso di Laurea Magistrale in Ingegneria Informatica

Routing and Caching on DHTs

Laureanda: Federica Bogo

Relatore: Prof. Enoch Peserico

A.A. 2009-2010

CONTENTS

1. INTRODUCTION

Currently, traffic generated by P2P systems has become a major portion of the Internet traffic, and it is still increasing. Due to the primary importance of this phenomenon, numerous studies addressing the problem of designing efficient P2P overlays have been proposed in the last few years.

The earlier proposed approaches offered schemes that nowadays are normally labelled as *unstructured P2P networks*: in these models, no precise control is held over object placement and flooding search protocols are mostly employed; whereas such networks generally provide good self-organization capabilities and are easy to manage in distributed and evolving environments, they may suffer in terms of scalability and load balancing. In order to address these problems, *structured P2P networks* (usually represented by *Distributed Hash Tables - DHTs*) have been introduced. Structured networks use specialized placement algorithms to assign responsibility for each object to specific nodes, and well defined directed search protocols to efficiently locate objects; moreover, they adopt ad-hoc strategies to improve load balancing among nodes, for example by using cryptographic hashes to spread the mapping between objects and locations.

However, such techniques generally lean on a problematic assumption: all the objects stored within the network are considered to have an equal *popularity* (that is, no object is supposed to be searched with higher frequency than others). In contrast, a real environment may present a very different situation: generally, relatively few highly popular objects are requested most of the times. If this more common skewed access distribution is not adequately handled, a heavy lookup traffic load can easily arise at the peers responsible for popular objects and at the intermediary nodes that route queries to those peers. In such a way, few individual nodes become easily overloaded, compromising the load balancing properties that DHTs want to achieve. The peak of this phenomenon is reached with the so called *hotspots*: hotspots take place when a large number of peers wishes to simultaneously access data from a very small set of nodes (or even from a single node), causing these target nodes to be swamped.

Note that, however, this skewed popularity could also bring some favorable opportunities: through years, various caching and replication mechanisms have been proposed, in order to achieve a substantial reduction of the search cost for popular objects and to balance the workload of the whole network. In our work, we will show that, despite all the countermeasures adopted through years for dealing with heavy bias and fluctuation in objects popularity, the problem of overloaded nodes can still arise in modern DHTs.

Interestingly, we identify the main causes of the persistency of this problem with two peculiar characteristics introduced in structured P2P networks: first, the substantial lack of flexibility in DHTs data placement and routing, that appear too strictly predetermined; second, the distinction between KBR (*Key Based Routing*) layer and "storing" layer. Routing schemes seem to be totally unaware of the content cached by any intermediate node chosen as "next hop": therefore, finding the requested value within an intermediary node's cache becomes a purely random variable.

After this "pars destruens", in which the main flaws of the existing protocols are depicted, a "pars construens" needs to be proposed; the challenge, here, is to design a novel, alternative overlay, which maintains a structured and efficient scheme but also introduces two features: a strict correlation between routing and caching (the latter must directly "guide" the former) and a randomized lookup algorithm. In a network of N nodes, we require that our overlay guarantees the following properties above all:

- each node experiences a load (i.e., number of received queries) $O(\log N)$;
- the size of each node's cache is kept $O(1)$;
- the complexity of finding the requested value (in terms of number of hops) is equal to $\log N$ or, at most, *poly*$\log N$.

Our aim is to develop a *robust* scheme, able to maintain those characteristics in *any* possible networking configuration, even an *unusual* and highly destructive one. Such situations can arise due to unexpected failures of nodes belonging to the network or, in particular, due to ad-hoc performed attacks. A simple example is a Denial of Service (DoS) attack that, generating an unanticipated, massive, rapid increasing in the number of requests targeting a limited and well defined set of nodes, causes these nodes to incur unbearably high loads, to dramatically degrade their performances and, eventually, to go down. Our P2P structure must take into account these anomalous conditions and develop adequate (and adaptive) mechanisms, in order to achieve robustness and resilience.

In this work, we'll propose and analyze two different caching algorithms for our overlay; in order to *formally* prove their robustness, we'll try to test them against an *adaptive adversary* that can choose the sender and the target of any issued query.

The rest of this work faces in a deeper and more articulate manner the analysis we've briefly outlined in this Introduction. In Chapter 2 we try to depict the "state of the art", presenting a survey of the most significant caching and replication schemes that have been proposed so far, for structured P2P networks and for unstructured ones as well; in Chapter 3, existing caching schemes are analyzed in order to find a lower bound on the number of *cache hits* that are expected to occur: the technique we employ applies to generic, simple models as well as to more sophisticated models (e.g., Kademlia-like structures). Chapter 4 presents our novel approach: the proposed overlay and the adopted routing and caching algorithms are explained in details, and their main qualitative and quantitative properties are formally defined. The theoretical analysis developed in Chapter 3 is then validated through the experimental results reported in Chapter 5. Finally, in the Conclusions we sum up our contributions and suggest some open problems.

2. ROUTING AND CACHING ON P2P NETWORKS: AN OVERVIEW

In the last few years, P2P networks have rapidly become a basis for building distributed applications. Generally, much of the attention has been focused on their construction and on the search efficiency issue: in particular, routing algorithms have been widely studied.

In most of the current search solutions, all peers are assumed to submit queries that uniformly search the contents stored in all nodes; unfortunately, the popularities of the contents are often quite skewed, making the workload over the whole network unbalanced. Therefore, additional mechanisms such as caching and replication have been introduced.

Initially, caching and replication schemes for P2P networks have been developed in order to handle bursts occurring in *web* traffic. Starting from the seminal work of Karger et al. [20], many caching algorithms for web environments have been proposed [39, 17]: in these cases, P2P-like structures are introduced in order to coordinate a distributed caching system and achieve load balancing for a small number of web servers.

In general, less attention has been paid to caching and replication mechanisms addressing exclusively P2P traffic: here, we'll summarize the main results achieved in this specific field.

After analyzing the skewed distribution of object popularity in greater detail, this chapter provides an overview of the most important P2P routing and caching algorithms proposed so far, for unstructured networks as well as for structured ones. The main purpose of this brief survey is to explore the *connections* that exist between routing and caching, in order to understand the limitations suffered by current approaches and develop a novel strategy.

2.1 Modeling P2P traffic

A deep understanding of the P2P traffic characteristics, especially of those relevant to caching, is a primary requirement in our study. Since the range of different aspects we could consider may be excessively wide and multifarious, our choice here is to analyze only the most significant one: the *distribution of object popularity.* The relative popularity of an object can be roughly defined as the probability of requesting that object relative to other objects. Clearly, object popularity is critical for the performance of any caching mechanism: intuitively, storing the most popular objects within nodes' caches is expected to yield higher hit rates than storing any other set of objects.

There is not an uniformly accepted pattern for representing the distribution of object popularity in P2P networks. In particular, two different models have arisen through years: the first is derived from the analysis of the traffic in unstructured networks, the second from the study of query distribution in DHTs. We briefly summarize both these schemes.

Mandelbrot-Zipf distribution The unstructured P2P traffic has been experimentally recognized as repetitive, predictable and so suitable to be cached even by the early measurement studies [21]. This observation has led to more articulated and abstract analyses: among them, two of the most exhaustive can be found in [13, 36]. While in [13] Kazaa[1] is quantitatively analyzed, the research in [36] focuses on Gnutella[2]; the results offered by both works are quite similar. In particular, the authors emphasize how the *Zipf* qualities commonly attributed to popularity distributions in the World Wide Web are not directly applicable to P2P networks. The main difference is well visible plotting the Zipf distribution and the P2P object popularity distribution on a *log-log scale* and not on a *linear scale*: in Figure 2.1, ranking based on object popularity versus number of requests issued for those objects is represented for Kazaa items as well as for WWW items: while the latter is well described by a Zipf line, the former does not seem to fit it adequately.

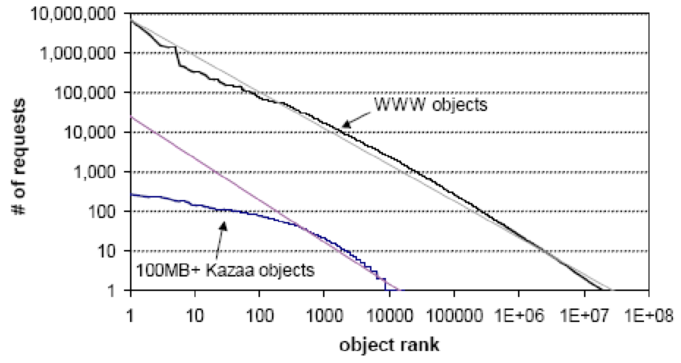

Fig. 2.1: Popularity distribution of Kazaa and WWW objects (curved lines) compared with the corresponding Zipf distribution (straight lines).

Indeed, the strongest difference is noticed among the most popular objects: a ”flattened head” is observable in the left-most part of the P2P popularity curve, while a straight line appears in the right-most part. The main reason of this phenomenon could be the difference among the clients acting in Web and those acting in P2P environments: Web clients often fetch the same Web page many times (”fetch-repeatedly”), whereas P2P clients rarely request the same object twice (”fetch-at-most-once”). The mathematical interpretation provided in [36] for this trend is summarized in a Mandelbrot-Zipf distribution. The Mandelbrot-Zipf distribution defines the probability $p(i)$ of accessing an object at rank i out

[1] kazaa.com

[2] Today, the term ”Gnutella” does not refer to any project or piece of software, but to an open protocol used by various clients, like Shareaza (shareaza.sf.net) and LimeWire (LimeWire.org)

of N available objects as:

$$p(i) \propto \frac{K}{(i+q)^{\alpha}}$$

where $K = \sum_{i=1}^{N} 1/(i+q)^{\alpha}$, α is the skewness factor and $q \geq 0$ is the so called *plateau* factor, responsible of the "plateau shape" near to the left most part of the distribution.

Zipf-like distribution A similar but simplified model seems to be widely accepted by research works focusing on structured networks: the experimental studies exposed in [4, 15] represent the popularity owned by objects in many DHTs by means of a simple *Zipf-like* distribution. In this way, the relative probability for a query requesting the object at rank i to be issued is defined as:

$$p(i) \propto \frac{1}{i^{\alpha}}$$

This model has an important implication [4], that deserves to be emphasized: with the random uniform placement of objects usually accomplished by DHTs and the Zipf-like selection that characterizes requested objects, not only the *request load*, but also the *routing load* resulting from the forwarding of messages along intemediary nodes exhibits powerlaw characteristics. As a consequence, an effective caching mechanism should focus on the target nodes and, in addition, on the path (or paths) covered for reaching those nodes.

While an outline of these models has been necessary in order to provide an adequate contextualization, it's enough for us to know that, generally, P2P networks deal with a small number of objects extremely popular, with a long tail of unpopular requests. It's not our aim to discuss in greater detail which approach between the two is more feasible. Indeed, we must keep in mind that, whereas the distribution exposed above is the most common, *unusual* (and destructive) scenarios can take place, due to unexpected failures of nodes belonging to the network or, more often, to ad-hoc performed attacks, like the Denial of Service (DoS) attacks. Clearly, a robust P2P structure must be able to handle these anomalous conditions in order to achieve robustness and resilience. We will extensively deal with these aspects in the following chapters.

2.2 Unstructured P2P networks

Early instantiations of well-known P2P networks, such as Gnutella, Kazaa and Freenet[3], represent the so called "unstructured networks". Such overlays are organized without a predetermined structure: every node can have an unbounded *degree* (i.e., number of neighbors) and queries are usually flooded widely through a large number of nodes. Usually, unstructured approaches have been labelled as the "first generation", implicitly inferior to the "second generation" structured algorithms we'll expose in the next section. Indeed, the broadcasting unstructured approaches present some flaws: they may have large routing costs, or even fail

[3] freenteproject.org

to find available content; despite these apparent disadvantages, however, this network topologies still offer interesting suggestions: this section aims to briefly review these features, in order to better evaluate if, in some cases, they could adapt to the context of DHTs.

2.2.1 Routing algorithms

Since, in these environments, the network does not have a well defined structure, routing algorithms are kept as simple and general as possible; every different algorithm simply defines how a query must be forwarded through nodes in order to discover the node storing the desired object. In general, proposed forwarding methods can be classified as follows [31, 23]:

Flooding A node issuing a query forwards it to all nodes adjacent to it (that is, to all its neighbors); then, if the nodes receiving the query do not have the requested object, they transmit the query to all their neighbors. This process is repeated until the requested object is eventually discovered. Each query has a TTL (Time To Live) value, which limits the maximum number of intermediate nodes through which the query can be forwarded; when the query is received by neighboring peers, the TTL value decreases by one and, when it becomes zero, the query is expired and deleted.

Expanding ring This approach is somehow similar to the previous one. A node first forwards its query as in the flooding method, setting a small TTL. If this query does not discover the requested object, the sender increases the TTL value and sends it out again. This process is repeated until the TTL value reaches some predetermined threshold.

k-Walker Random Walk The aim of this approach, with respect to the previous ones, is to reduce the traffic generated by query messages. The requesting node generates k queries and each of them is transmitted to a randomly chosen adjacent peer at each step; for each query a TTL is set. We briefly mention two specializations of this approach, HDF (High Degree Forwarding) and LDF (Low Degree Forwarding): according to these techniques, queries are forwarded no longer to totally random neighbors, but to peers with a high degree and a low degree, respectively; in this way, the k-walker random walk approach tries to take into account the characteristics of the network.

2.2.2 Replication and caching approaches

In unstructured networks, a lookup process can reach an excessive number of nodes and, moreover, can fail to discover objects that are available; several unnecessary duplicate messages may be sent, since the number of nodes visited per round increases exponentially with the degree. In order to solve this problem, caching and replication seem to be attractive solutions: indeed, various approaches have been proposed, primarly in order to reduce search latency and improve object availability [23, 6, 41]. We briefly outline the most important caching and replication techniques.

Uniform, proportional and square root replication In the following, q_i will refer to the popularity of object i (in terms of number of requests issued for it); r_i indicates the number of nodes i is replicated in; the total number of objects stored in the network is R and the number of nodes is m ($\sum_i^m r_i = R$). According to the *uniform* strategy, replications are uniformly distributed throughout the network:

$$r_i = \frac{R}{m}$$

This means that, for each object, approximately the same number of replicas is created: while this controls the overhead of replication, replicas may be found in places where objects are not accessed by any node. A more "well-judged" approach is offered by the *proportional* replication, whereby the number of copies for each object is proportional to its query distribution:

$$r_i \propto q_i$$

Consequently, the higher the query rate of an object, the higher is the number of copies for that object; here, the drawback is that, although queries on popular data are processed efficiently, unpopular objects lookups may take an unacceptable long time. In the case of *square-root (SR)* replication, the number of replicas of an object i is proportional to the square root of the query distribution q_i:

$$r_i \propto \sqrt{q_i}$$

It has been shown [23, 6] that optimal replication is achieved exactly when the number of replicas per object is proportional to the square root of the object's popularity. In [41], the most feasible eviction policy for cache items is studied too. *LRU (Least Recently Used)* technique seems able to achieve near-optimal replicas distribution, exhibiting also good properties of convergence in transient environments.

Owner replication, path replication and random replication In contrast with the previous techniques, some schemes have been proposed [23], that replicate an object only when a query successfully retrieves it: this approach could be considered in some way as "dynamic", while the previous one is "static" since it does not take into account the difference between successful and unsuccessful queries. According to the *owner* replication, an object is replicated only at the requesting node, in such a way that the number of replicas for an object increases in proportion to the number of (successful) requests issued for it. Instead, the *path* replication creates copies of an object on all nodes along the path from the providing node to the requesting. Finally, *random* replication distributes the replicas in a random order rather than following the topological order; in particular, adopting k-walkers random walks also for replication purposes, random replication scheme seems a good approach for achieving both small search delays and small deviations in search.

2.3 Structured P2P networks

Theoretically, P2P networks should be able to easily grow, reaching a very great number of partecipating nodes; for achieving this, high scalability is required, mainly in terms of search performance: such a requirement translates into minimizing the number of hops a message must cover in order to satisfy a query, and this is, in effect, the aim of structured systems, where the overlay network is constructed using a deterministic procedure. Such networks rely on a simple mechanism: every object is assigned a key (e.g., the file name); keys and nodes (most of the time, nodes are represented by their IP addresses) are mapped into a common *identifier space*, usually by means of a *Distributed Hash Table (DHT) abstraction*. Given a key, the node that currently has references to the corresponding object is the one whose hashed value is closest to the key's hashed value. Thanks to this organized structure, any object can be located within a bounded number of routing hops, using a small per-node routing table. Besides these DHT-based schemes, randomized P2P networks like Symphony, SkipGraph, and more organized overlays like Baton [31] are all examples of structured P2P overlays too. Similarly to what we've done in the previous section, we survey the most important routing strategies employed in DHTs and some caching and replication schemes adopted. A deeper knowledge of these aspects is necessary in order to better contextualize our work, in which an innovative DHT protocol is proposed.

2.3.1 Distributed Hash Tables

As we've briefly anticipated, a Distributed Hash Table (DHT) is a structured overlay network protocol that provides a decentralized search service, mapping objects to peers. Indeed, the lookup operation it implements is similar to a *hash table*: objects stored in the network are seen as $< Key, Value >$ ($< K, V >$) pairs; any participating node can efficiently retrieve the value associated with a given key, since the information about the network topology is maintained using a data structure distributed among all participants. Thanks to this data structure, the routing procedure is more efficient, not completely "blind".

In general, DHTs form an infrastructure that can be used to build more complex services, not only P2P file sharing systems but also cooperative web caching, multicast, domain name and instant messaging services. However, regardless the particular context in which they may be utilized, DHTs require an accurate definition of two different aspects: the *mapping* of keys and nodes into a common identifier space and the strictly interconnected notion of *metric*. In particular, the metric is used to establish a relationship between any pair of nodes, in such a way that their *distance* can be univocally calculated (note that the notion of distance refers to the identifier space, and is totally unrelated to physical considerations).

Let us consider in greater detail the first aspect, that is the mapping into a well defined identifier space. First of all, an appropriate, abstract space must be defined by every DHT implementation: most of the time, it corresponds to the set of n-bit strings (where n normally is equal to 160, with a total of 2^{160} available *addresses* or *IDs*). Keys and nodes are then mapped into this common space, and a partitioning scheme splits ownership of different identifier "zones" among all the participating nodes. In particular, every node is responsible for the keys which are closest to it, that is, which are mapped in its "zone".

The mapping is usually accomplished by means of a *consistent hashing* function [20]. Consistent hashing has the essential property that removal or addition of one node changes only the set of keys owned by the nodes with adjacent IDs, and leaves all other nodes unaffected. In contrast, in a traditional hash table, addition or removal of one node would cause nearly all the keys to be remapped. Since any change in ownership typically corresponds to bandwidth-intensive movement of objects stored in the DHT from one node to another, minimizing such reorganization is required to efficiently support high rates of churn. Furthermore, consistent hashing is supposed to produce values that are equally spread across the ID space, so that the responsibility for the keys and the subsequent load are distributed among all peers in a fair manner: if there are M objects to store and N nodes (usually, $M >> N$), every node is responsible for about M/N resources; as a consequence, a satisfactory load balancing among all peers is achieved.

An overlay network then connects the nodes, allowing them to find the owner of any given key across the identifier space. Each node maintains a set of links to other nodes (its *neighbors*) in a structure called *routing table*: such neighbors are usually located at different distances, so that a partial but wide knowledge of the entire network can be held by each node.

A node picks its neighbors according to a certain structure, that represents the network's *topology*: together, these "neighborhood" links form the *overlay network*.

With this overlay as a basis, lookup, store and update operations are efficiently performed. In particular, it is over the routing procedure that all other operations are built: as we'll see in the following subsection, storing and updating start with a certain ID search. So, in a DHT, the routing algorithm must be accurately developed.

When looking for any key k, each node either is assigned a nodeID which owns k or has a link to a node whose node ID is closer to k, in terms of identifier space distance. It is then easy to route a message to the owner of any key k using the following greedy algorithm (note that it is not necessarily globally optimal): at each step, forward the message to the neighbor whose ID is closest to k. When there is no such a neighbor, then the message must have arrived at the closest node, which is the owner of k as defined above. This style of routing is sometimes called *key-based routing (KBR)*. Beyond basic routing correctness, two important characteristics are required: the maximum number of hops in any route (*path length*) should be low, so that requests complete quickly; the maximum number of neighbors of any node (*node degree*) should be low too, so that the maintenance overhead is not excessive. Of course, having shorter routes requires higher maximum degree. Some common choices for maximum degree and path length are the following (N is the number of nodes in the DHT):

- degree $O(1)$, path length $O(N)$;
- degree $O(\log N)$, path length $O(\log N/\log\log N)$;

- degree $O(\log N)$, path length $O(\log N)$.

The third choice is the most common, even though it is not optimal in terms of degree/route length tradeoff, because such topologies typically allow more flexibility in the choice of neighbors. Many DHTs use that flexibility to pick neighbors which are close in terms of latency in the physical underlying network. Note how the maximum path length is closely related to the overlay's *diameter* (i.e., the maximum number of hops in any shortest path between nodes). Clearly, the network's path length is at least as large as its diameter, so DHTs are limited by the degree/diameter tradeoff which is fundamental in graph theory; however, path length can be greater than diameter, since the greedy routing algorithm may not find the shortest paths.

2.3.2 Routing algorithms

This subsection presents some interesting DHT overlays developed during the last few years. Before starting our survey, let us synthetize the ideas exposed in the previous subsection and point out, once more, how DHTs should provide four favorable characteristics [31]:

- Low degree: each node keeps only a small (and bounded) number of connections to other nodes (its neighbors);
- Low diameter: the number of intermediate hops and the delay introduced by the indirection of the lookup process should be as small as possible;
- Greedy routing: nodes independently calculate a short path to the target and, at each hop, the query moves closer to the target;
- Robustness: a path to the target should be found even when one or more links or nodes fail.

In order to achieve these goals, an efficient and robust routing algorithm must be proposed, as well as an underlying well defined structure; indeed, every structured network must take into account both these aspects:

- **Structuring**: the organization of the *topology* of the network (i.e., how nodes are interconnected) and, also, the data structures maintained at each node and used for locating and routing; briefly, the *overlay*;
- **Routing**: the algorithms that, on the basis of the underlying structure, are used to locate an object and to route a query to it. Routing procedures are implemented using an iterative greedy algorithm that, at each hop, decreases the distance between the destination node and the current intermediate node.

Since these apects are inseparable when describing a DHT, in the following we'll expose the most known overlays taking into account both of them.

PRR In [26], Plaxton, Rajaraman and Richa (PRR) propose one of the early examples of DHT. Objects and nodes are assigned a simple ID (tipically, a 160 bit long string). Every node is seen as the root of a *spanning tree*: a message routes toward an object by matching longer address suffixes, until it encounters either the object's root node or another node storing a "nearby" copy. The main disadvantages of this scheme are the requirement of a global knowledge in order to construct the overlay and the single point of failure constituted by every object's root node. Furthermore, note that the network is assumed to be static (that is, nodes can not join or leave the network) and no mechanism is adopted in order to avoid traffic congestion occurring at popular nodes.

Pastry Each Pastry [34] node has a unique, uniform randomly assigned nodeID in a *circular* $128 - bit$ identifier space. For the purpose of routing, nodeIDs and keys are thought of as a sequence of digits in base 2^b (where b is a configuration parameter). A node's routing table is organized into $128/2^b$ rows and 2^b columns; the 2^b entries in row n of the routing table contain the IP addresses of nodes whose nodeIDs share the first n digits with the owner node's nodeID; the $(n+1)th$ nodeID digit of the node in column m of row n is equal to m. The column in row n that corresponds to the value of the $n+1$'s digits of the local node's nodeID remains empty. Since the uniform random distribution of nodeIDs should ensure an even population of the identifier space, on average $\log_{2^b} N$ levels are populated in the routing table. Each node also maintains a *leaf set*: this represents the set of l nodes with the nodeIDs numerically closest to the present node's nodeID, with $l/2$ larger and $l/2$ smaller nodeIDs than the current node. During the message routing, at each hop a node seeks to forward 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 current node's shared prefix. If such a node can not be found, the message is forwarded to a node that shares a prefix with the key at least as long as the present node, and is numerically closer to the key than the present node's nodeID: such a node is guaranteed to exist in the leaf set unless the message has already arrived at the node with numerically closest nodeID or at its immediate neighbor.

Chord In Chord [40], each node and each object are assigned an m-bit ID: while the nodeID is obtained by hashing the node's IP address, the objectID is calculated associating each object with a unique key K (that is, every object is considered as a key-value pair $< K, V >$) and hashing the key itself into an ID. All the possible $N = 2^m$ nodeIDs are ordered along a one-dimensional virtual circle (*ring*), into which nodes are mapped according to their nodeIDs. For each nodeID, the first peer located on its *clockwise* side along the ring is called its *successor node* (*succ(nodeID)*). Similarly, the objectIDs cover a virtual position in the circle too, namely the position P obtained by hashing the corresponding key ($P = hash(K)$). Thanks to this mechanism, assignment of objects to nodes can be decided. If $P = hash(K)$, the object $< K, V >$ is stored in the first node on P's clockwise side, that is called the *successor node* of P (*succ(P)*). To route efficiently, each node contains part of the mapping information in its *finger table*.

In the view of each node, the virtual circle is partitioned into $1 + \log N$ segments: itself and $\log N$ segments with length 1, 2, 4,..., $N/2$; every node maintains $\log N$ entries in its table: each entry contains the information for one segment, namely the boundaries and the successor of its first virtual node (see Figure 2.2). In this way, each node uses $O(\log N)$ memory to maintain the topology information. When a node wants to retrieve the object $< K, V >$, it first calculates $P = hash(K)$. Using the finger table, the successor of the segment that contains P is selected to be the next "router", until P lies between the start of the segment and the successor (this means that the successor is also P's successor, i.e. the target). The distance between the target and the current node decreases by half after each hop, providing a routing time $O(\log N)$.

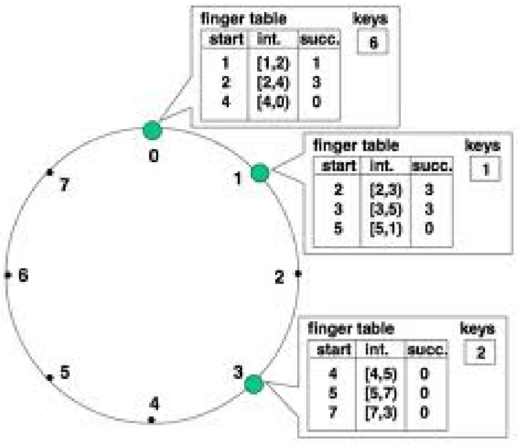

Fig. 2.2: The ring-like structure of Chord. Each node stores part of the routing information within its finger table.

CAN Content Addressable Network (CAN) [30] maps the IP address of each node and the unique key associated with every object into an identifier space by means of a hash function too. Unlike the Chord's ring topology, CAN maintains a d-dimensional *Cartesian coordinate space*, organized on a "d-torus": through the hashing, nodes and objects are assigned a d-dimensional vector, which can be considered as a point in the coordinate space. This space is partitioned into many small d-dimensional *zones*: each node corresponds to one of such zones (namely, to the zone containing the IDs that are closer to the node itself than to any other node) and stores the objects that are mapped there by the hash function (see Figure 2.3). Two nodes are neighbors if their coordinate spans overlap along $d - 1$ dimensions and differ in one dimension. For locating a key, its virtual position is first calculated; then, starting from the requesting node, the query is passed through the neighbors until it reaches the target machine: in particular, a greedy algorithm is employed, according to which the neighbor whose coordinates are most similar to those of the target is chosen. In such a d-dimensional space, each node maintains $O(d)$ neighbors and the routing path length counts $O(d(n^{\frac{1}{d}}))$ hops. Unlike Pastry and Chord, CAN's routing table does not grow with the network size, but the number of routing hops grows faster than $\log N$ in this case.

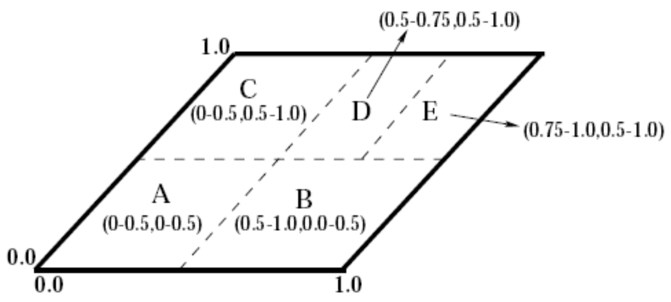

Fig. 2.3: A 2-dimensional Cartesian coordinate space subdivided into five zones.

Kademlia Since Kademlia [24] will be an important subject of study in our work, we'll expose its main features in greater detail in Chapter 3.

2.3.3 Replication and caching approaches

DHTs have rapidly become a basis for building distributed applications in the last ten years, but, although close attention has been focused on their construction, their efficacy is still under scrutiny. In particular, $O(\log N)$ hop long routing paths could introduce a latency that is unacceptable for some applications (e.g., DNS servers); furthermore, a balanced distribution of load among all the peers, that is widely claimed in theory, is not guaranteed in practice. In order to remedy, structured networks adopt replication and caching mechanisms too. Since a wide range of available techniques exists, we introduce a classification depending on four parameters [28]:

- Replication typology: replication can be *proactive* (replicas of an object are cached by nodes regardless of the queries issued for that object) or *passive* (there is caching only for those contents that have been already requested);
- Placement strategy: this aspect determines where the replicas should be placed (for example, on all the nodes along a lookup path, near the destination node, at the node that requested the object or purely randomly);
- Replication degree: it defines how many replicas should be created for an object;
- Eviction policy: when an item is requested to be inserted into a full cache, it must be clear which value already stored, if any, should be removed.

Considering these different aspects, we can go on and analyze some remarkable examples.

EAD *EAD (Efficient and Adaptive Decentralized File Replication)* approach [38] presents two main purposes: first, replicating objects in such a way that replicas are fully utilized and, second, removing under-utilized replicas, possibly taking into account also the time-varying character of object popularity. In order to do this, two different typologies of nodes are chosen as "*replica* nodes" (that is, nodes that must store the replica of a given object): *frequent requesters* and nodes that carry more query traffic load ("*traffic junction* nodes"), as shown in Figure 2.4. Every node calculates a *query rate* q_o for each object o, as the number of queries for o initiated or forwarded by the node itself during a unit time; when an object requester's q_o exceeds a predetermined threshold T_o, the node adds a replica request into the query, in such a way that the node holding the object sends it back to the replica requester. The value of q_o for all the cached objects is periodically checked: when there are few queries for one of those objects, the value of q_o decreases below T_o and the corresponding replica is removed.

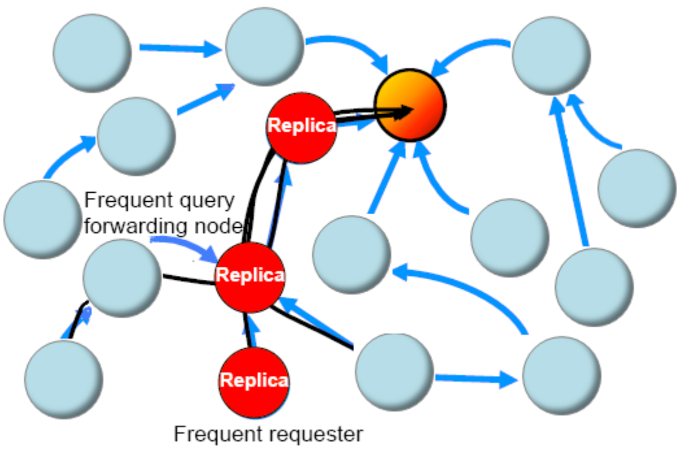

Fig. 2.4: In EAD, frequent requesters and traffic junction nodes become replica nodes.

HSCM In [5], Chen et al. describe the so called *Hot Spots Caching Mechanism (HSCM)*. Although this caching approach could be theoretically used together with any type of structured network, the authors combine it with an hybrid-Chord system: in such a network, k Chord rings are overlayed one on top of the other, in order to improve lookup efficiency and robustness. Since, in Chord, the node responsible for an object $< K, V >$ is its successor node (*succ(P)*, where, as above, $P = hash(K)$), the value HSCM eventually caches is the $< K, IP(K), Port(K) >$ tuple, where the last two values denote, respectively, the IP address and Port number of the successor node. In this way, a cache hit lets the querying node jump directly to *succ(P)* in one only more hop. The placement strategy consists in a simple path replication, with an eviction policy guided by the LRU mechanism. The LRU algorithm implemented by the requesting node is different from that implemented by the intermediate nodes along the lookup path: if a cache hit has occurred, both kinds of node change the priority of the queried key, increasing it; if a cache miss has occurred instead, the lookup process reaches the actual successor node: the requesting peer uses the lookup result and replaces the key holding the lowest priority within its cache with the new value, while intermediate nodes do nothing. The key idea of this (passive) approach is that the more popular is the key, the longer is the time it will stay stored within the cache: as a consequence, popular keys should not be replaced easily.

LAR The approach proposed in [19], *LAR*, is a little more complicated. Indeed, the sender node could eventually store one of more replicas of different objects, assigned to it by the nodes encountered along a search path. In order to do this, each node is equipped with additional advanced functions: whenever a node receives a query (whose it could be the target or not), it checks which is more heavily loaded among the sender peer and itself, and allocates replicas of some of its own data if it is more congested. Furthermore, in order to decrease the number of search hops, cached data are distributed over the P2P network. A cache entry contains the ID of an object and a reference to the node storing it, and is distributed over the network in a random fashion, usually piggybacked onto the search queries: this can be seen as a type of proactive caching.

Last hop caching The technique of Bianchi et al. [4] represents a passive scheme, and differs from the previous ones mainly for the replica placement adopted: indeed, a replica is stored at the *last node* (different from the target node) along the lookup path. This mechanism should offer the advantage that an object is cached in the neighborhood of its owner, where the possibility for a query to hit a replica is much bigger than elsewhere in the system (due to the convergence property usually owned by DHTs [14]). In particular, given an object, the node chosen for caching it is the one that most frequently served as last hop for this object, considering all the followed lookup paths. For determining when an object has to be cached, each node maintains a *per-object counter* (it counts up the number of received queries for each object held by the node) and a *per-node counter* (it considers the total number of received requests). Each time a defined per-node counter threshold is reached, a weight is computed for each object held by the node, based on the per-object counters: the most popular object on the node is the object with the highest weight, and this is the value that will be cached. In particular, to compute the popularity of an object, a weighted moving average is used, where the weight is computed as a combination of its previous value and the value computed over the last period. For the eviction policy, whenever a caching request is received and the storage capacity is exhausted, the replica entry with the lowest weight is discarded.

Beehive Beehive [27] is a general replication framework able to operate on top of any DHT that uses prefix-routing, providing an important example of proactive caching. It applies a simple replication mechanism: an object is replicated at *all* nodes k hops distant (or lesser) from the owner node, in order to reduce, correspondingly, the lookup latency by k hops. The extent of replication in the system, potentially excessive, is controlled by assigning a *replication level* to each object: an object at level i is replicated on all nodes that have at least an i digit long matching prefix with the object; in this way, queries requesting objects replicated at level i incur a lookup latency of at most i hops. According to this concept of replication level, in a network of size N, objects stored only at their owner nodes are at level $\log N$, while objects replicated at level 0 are cached at all the nodes in the system. Then, the main problem is finding the appropriate amount of replication for each object based on its popularity: in other words, the goal of Beehive's replication strategy is finding the minimal replication level for each object such that the average lookup performance for the system is represented by a constant number (C) of hops. Beehive is built on the consideration that queries usually follow a power law distribution, so all the nodes continuously monitor the network in order to find a good approximation for this distribution. First, during an analysis phase, every node, based on its measurements of the received queries and on the information obtained from a limited group of neighbors, estimates the popularity of every object and, using also the latest calculated distribution parameter, gets a new estimate for it; then, according to this result, nodes locally change the replication levels of every object. These values are stored within each node's routing table. Later, during the replication phase, each node is responsible for replicating an object on other nodes at most one hop away from itself (namely, at nodes that share a prefix one digit shorter than the current node). Whereas it seems an ingenious technique, the complex replication mechanism each node must periodically accomplish increases the time required for refreshing the whole system and lessen the network's scalability.

Erasure coding All the techniques exposed so far aim to keep a certain number of available replicas for certain objects, establishing where they must be created and how long they must be kept. In contrast, with an *erasure coding* redundancy scheme [32], each object is divided into m fragments and recorded into n fragments which are stored separately ($n > m$). The key property of erasure codes is that the original object can be reconstructed from any m fragments, where the combined size for the m fragments is approximately equal to the original object size. While this approach reduces the consumption of storing resources, it seems to be hardly usable as caching mechanism in P2P environments. First of all, it introduces complexity in the system, mostly due to the necessity of encoding and decoding different data blocks; furthermore, it could be difficult to maintain data consistency: the impact of data mutability on the efficiency provided by this mechanism should be carefully evaluated.

The schemes just described are abstract caching mechanisms, adaptable to any DHT that does not have any in-built mechanism for dealing with non-uniform query distributions. Now, we briefly mention two (complete) distributed file sharing applications, PAST [33] and CFS [7], that naturally incorporate their own distributed replication scheme: indeed, both reserve a part of the storage space at each node to cache specific query results, in order to improve subsequent search performance; furthermore, they try to maintain a constant number of replicas of each object, for enhancing the system's fault tolerance.

CFS *Cooperative File System (CFS)* is a P2P read-only storage system; it presents stored data to applications through an ordinary filesystem interface, where servers store uninterpreted blocks of data with unique identifiers (that is, servers provide block-level storage) and clients retrieve blocks from the servers and interpret them as file systems. The core of the CFS software consists of two layers, *DHash* and *Chord*. The DHash layer implements a DHT for block storage, so performs block fetches for the client, distributes the blocks among the servers, and maintains cached and replicated copies. The Chord's distributed lookup algorithm is then used to locate the servers responsible for a block. The mechanism of caching used by CFS, purely passive, is quite simple and resembles the path replication: blocks are cached in every node *along the path* from the requesting client to the server storing the desired block. Once the initiating node has found the block, it sends a copy to each of the servers it contacted during the lookup; these servers add the block to their caches. This caching scheme is expected to produce high cache hit rates because the lookup paths for the same block from different sources will tend to intersect as they get closer to block's successor server; specific problems related to overloaded servers or hotspots are not taken into account.

PAST In CFS, blocks of data are stored: while this approach permits parallel blocks retrievals, it may increase the file retrieval overhead, as each distinct block of the file must be located using a separate Chord lookup; we find an antithetical condition in PAST [33], a P2P persistent storage utility built over Pastry: indeed, PAST handles files as a whole, not blocks.

A subtle distinction is introduced between replication mechanisms and caching mechanisms. The replicas of a file are stored on the k nodes that are numerically closest to the objectID corresponding to the file itself (k can be no larger than $(l/2)+1$, where l is the size of the leaf set); these replicas are maintained primarily for reasons of availability, rather than for reasons of load balancing and latency reduction: a highly popular file may demand many more than k replicas for bearing its lookup load while minimizing client latency. In such cases, caching helps. PAST nodes "recycle" the unused portion of their advertised disk space to cache files; in this way, as the storage utilization of the system increases, cache performance degrades gracefully. The eviction policy is based on the *GreedyDual-Size (GD-S) policy*, which was originally developed for caching web proxies. Upon insertion or use, the weight H_d associated with a file d is set to $c(d)/s(d)$, where $c(d)$ represents a cost function associated with d, and $s(d)$ is the size of the file d. When a file needs to be replaced, the file v is evicted whose H_v is minimal among all cached files. Then, H_v is subtracted from the H values of all remaining cached files. If the value of $c(d)$ is set to one, the policy maximizes the cache hit rate.

2.4 Conclusions

The outline we've depicted in this chapter does not aim to be a detailed and exhaustive study of the main research works that has been written on P2P networks during the last ten years; for such a purpose, the reader can follow the valid bibliographic references we provide. In contrast, this brief "sketch" has been introduced mainly in order to emphasize two important concepts: first, the relationship existing between structured and unstructured networks; second, the strict interdependence that couples routing and caching mechanisms.

In same regards, considering what we've exposed, it's lecit to cast unstructured and structured algorithms as *competing* alternatives, introducing a clear distinction between the "first generation" of unstructured approaches and the "second generation" of structured algorithms; after all, this opinion is quite common and usually accepted. However, note how both those approaches offer some advantages, but also suffer a certain number of disadvantages. When generic key lookups are required, the structured routing schemes can guarantee location of the target ID within a bounded number of hops, while the broadcasting unstructured approaches may have large routing costs or even fail to find available content. On the other hand, there have been two main criticisms of structured systems: first of all, it is not clear how well DHTs can support join and leave of nodes, maintaining each node's routing table in a "consistent" state; since P2P systems often exhibit *churn*, with peers continually arriving and departing (or even failing), the resilience of the network under these conditions must be proved. Another aspect concerns keyword searches: structured systems don't support them, and, in general, complex queries as well as unstructured systems.

Maybe, a good idea is to consider unstructured and structured proposals as *complementary*, not competing, summing up the values of both schemes. In order to justify this approach, one can consider how most unstructured schemes have evolved, incorporating, somehow, a defined structure. One example of this phenomenon is the Gnutella network: for improving the scalability of the system, a special kind of hierarchy has been introduced among nodes, according to which peers are either ultrapeers or leaf nodes; this hierarchy is augmented with a query routing protocol whereby ultrapeers receive a hashed summary of the resource names available at leaf nodes.

The approach we'll delineate in the following chapters is specular to that phenomenon: in other words, we'll exactly follow the opposite direction, introducing some "tipically unstructured" features into our novel DHT. No inefficient flooding mechanisms or hierarchic schemes will be utilized: simply, we'll see how, in some cases, randomized lookup paths can help increasing the search performance and decreasing the load experienced by some nodes.

Another crucial point will be, as we've anticipated, the relationship between caching and routing. All the techniques we've reviewed so far seem to keep a well defined distinction between these two aspects: routing and caching are located at different levels, since the former is related to search, the latter to storage. The main problem suffered by this approach is that request and routing load can not be adequately balanced through caching, and caching can not be fully utilized through routing. Our work aims to show that, as long as the occurrence of a cache hit is a purely random variable, no satisfactory load balancing is guaranteed, nor the risk of swamped nodes is averted, especially in case of DoS attacks. In the following chapters, after analyzing these aspects in details, we'll propose a valid solution that tries to bring together both routing and caching strategies.

3. CACHING EFFECTIVENESS ANALYSIS FOR STRUCTURED P2P NETWORKS

In this chapter, we'll analyze the traditional caching approaches in greater detail, in order to show their substantial *ineffectiveness* in a very generalized context.

After exposing the basic model of structured network we'll adopt and some mathematical prerequisites, we provide four distinct properties a routing algorithm must exhibit in order to be subject to our analysis: as it will be clear, when all these four conditions hold at the same time in a structured P2P network, the worst case per-node load is (asymptotically) the same that would be obtained if no active caching mechanism were accomplished by any node in the network.

Finally, some considerations about the general validity and the consequences of our results are carried out.

3.1 Generalized model

In our study, we'll adopt a generalized structured network model. While such a model is necessary for carrying out a clear theoretical analysis, an excessive simplification could undermine the applicability of the subsequent results. As we'll explain in greater detail in section 3.5, the assumptions we'll adopt don't invalidate the applicability of our results to a wide range of real contexts. Note also that, even if, in some cases, we need to rely on a particular DHT structure for making our examples, the only actual requirements for our analysis to hold are those depicted in section 3.2; in fact, the reader could apply our analysis to the DHT structure he is most familiar with, without affecting its validity.

First of all, we assume that our network has an identifier space of size N, and that there exists a *bijective correspondence* among IDs and nodes (i.e., our network is *full*, completely populated: every node is assigned exactly one key). For keeping our model as simple as possible, no particular assumptions regarding the homogeneity (or the heterogeneity) of the N nodes will be made; the *load* experienced by a node is simply represented as the number of requests that reach it, without any reference to specific physical resources' consumption (e.g., available bandwidth or CPU cycles). The only feature we'll assume to be equal for all the network's nodes is their *cache size*; however, this restriction could be easily removed considering as our "cache size" the greatest value available among all the nodes.

Two specific subsets of nodes will be particularly important in our analysis; a formal definition is provided for both of them:

Definition 1. *If K is the set of popular keys k_i ($0 \leq i < \kappa$, where $\kappa = |K|$), the* hot zone M *is the set of all the nodes* m_i *that store a popular key* k_i*; we assume that* M *is a proper and not empty subset of the entire network.*

Definition 2. *The* outer border E *is the set of all the nodes* e_i *that don't belong to* M*, but have at least one neighbor belonging to* M*.*

For the sake of clarity, we provide a list of the abbreviations that will be used in our analysis:

- μ: hot zone's size (that is, $\mu = |M|$). Note that, in our "full network" model, the hot zone's size corresponds to the number of popular keys κ;
- D: upper bound on the number of hops needed to reach a node from any other node in the network, according to the adopted routing algorithm;
- c: cache size of each node. In some cases, the cache size c may be expressed as a function of μ and D:

  $$c = \frac{\mu}{qD}$$

  for an appropriate integer q, $q \geq 1$. Conversely, we could also express the hot zone's size μ as

  $$\mu = qcD$$

A fundamental assumption is that the nodes belonging to the hot zone are mapped into a *continuous interval* of IDs; in other words, we are requiring that, if popular keys are ordered by increasing ID and the popular key k_0 is mapped into ID x, then key k_1 is mapped into ID $x+1$ and so on, until we reach the key $k_{\kappa-1}$ corresponding to ID $x+\kappa-1$. Without loss of generality, we may consider $x = 0$. Figure 3.1 provides an example of such a hot zone in a tree-like scheme that represents a network of size 8: here, each network's node corresponds to a different leaf and, following the labels placed on the arcs along the path from the root node to a leaf node, the ID of the corresponding leaf node can be derived.

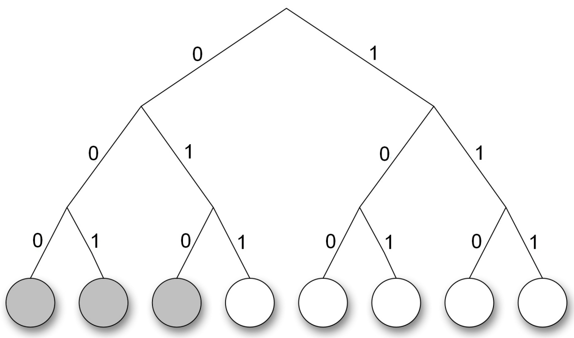

Fig. 3.1: A tree-representation of a network of size 8; the filled nodes represent a hot zone of size $\log N = 3$.

The importance of this (restrictive) assumption will be discussed in greater detail in section 3.5.

3.1.1 Mathematical background

In the following, we will make use of some fundamental probabilistic tools [25], briefly summarized in this subsection.

First of all, we introduce the *Chernoff bounds for the binomial distribution.* Recall that a random variable has a binomial distribution $B(n,p)$ if

$$Pr(X=j)=\binom{n}{j}p^j(1-p)^{n-j}, 0\leq j\leq n$$

The Chernoff bounds limit the probability that a random variable falls too far from its expectation. The following theorem gives Chernoff bounds for the binomial distribution.

Theorem 1. *Let X be a random variable B(n,p). Then, for any $\delta > 0$, the following Chernoff bounds hold:*

$$Pr(X\geq(1+\delta)np)<\left(\frac{e^{\delta}}{(1+\delta)^{(1+\delta)}}\right)^{np}$$

$$Pr(X\leq(1-\delta)np)<\left(\frac{e^{-\delta}}{(1-\delta)^{(1-\delta)}}\right)^{np}.$$

Furthermore, if $\delta\in(0,1]$, the above expressions can be simplified as follows:

$$\Pr\left(X\geq(1+\delta)np\right)\leq e^{-np\delta^2/3}$$

$$\Pr\left(X\leq(1-\delta)np\right)\leq e^{-np\delta^2/2}$$

Combining the two bounds, we get the inequality

$$\Pr\left(|X-np|\geq\delta np\right)\leq 2e^{-np\delta^2/3} \tag{3.1}$$

.

In order to construct our proofs, we'll make use also of a simple lemma [25].

Lemma 1. *Let $X_1, X_2, ..., X_n$ be a sequence of random variables in an arbitrary domain, and let $Y_1, Y_2, ..., Y_n$ be a sequence of binary random variables, with the property that $Y_i = Y_i(X_1, ..., X_{i-1})$. If*

$$\Pr(Y_i=1|X_1,...,X_{i-1})\leq p$$

then

$$\Pr\left(\sum_{i=1}^{n}Y_i\geq k\right)\leq Pr(B(n,p)\geq k)$$

and similarly, if

$$\Pr(Y_i=1|X_1,...,X_{i-1}\geq p)$$

then

$$\Pr\left(\sum_{i=1}^{n}Y_i\leq k\right)\leq\Pr(B(n,p)\leq k)$$

.

3.2 Routing assumptions

This section introduces the four properties a routing algorithm must satisfy in order to be subject to our analysis; as it will be shown in section 3.5, these conditions are all necessary for our proof to hold.

Property 1 (Diameter property). *An upper bound for the value of D (recall that D represents the maximum length of a lookup path) can be directly derived from the adopted overlay and the implemented routing algorithm (usually, $D = \log N$, where, as above, N is the size of the identifier space).*

Property 2 (Distance minimization property). *At every hop along the path from the node issuing the request to the target node storing the desired key, the distance between the current node and the target node is strictly less than the previous one.*

Property 3 (Prefix-guided path property). *Consider the hops covered by a lookup path requesting a popular key, starting from the sender node n, until an outer border's node is reached; this "first part" of the path could be (correctly) traveled, regardless of which popular key is actually requested; roughly speaking, if $\mu = 2^l$ $(0 < l < \log N)$, an outer border's node reached after at most* $\log N - l - 1$ *hops is a valid intermediate hop for reaching any popular node.*

Property 4 (Caching-independent routing property). *At every step along the lookup path, the "next hop choice" is totally* independent *from caching considerations.*

It can be easily seen that all these properties are implicitly or explicitly assumed by every DHT routing algorithm exposed in the previous chapter.

3.3 Lower bound analysis

Our lower bound analysis is performed comparing a totally generalized caching algorithm against an adversary model. After defining this model in details, we'll apply our analysis to three slightly different routing algorithms.

Note that, in the following subsections, a generic terminology is used; the translation into specific, different DHTs should be straightforward. Using a Kademlia-like terminology, we refer to the number of neighbors every node stores within its routing table, in corrispondence with every possible "distance", as the *bucket's size*. For example, if we consider that routing tables are organized by means of 1-buckets, for every distance only one neighbor is provided; instead, in the case of 2-buckets, two neighbors are known, and so on.

As we've anticipated, three proofs are provided; each one refers to a different scenario. A brief summary of the three cases we'll analyze helps us introduce the terminology that will be adopted in the following subsections:

- *1-buckets network model*: in this network, each node has a routing table with buckets of size 1; this simplified model lets us point out the key idea at the basis of our analysis;

- *k-buckets network model*: the approach developed for the previous case can be easily extended to a more general and common model, where each node's routing table is organized by means of k-buckets (k represents a parameter);
- *Kademlia-like model*: finally, introducing some adjustments in our argumentation, we'll be able to show how our results are still valid even for a specific case study, namely the Kademlia network; in particular, an interesting problem we'll face in details in our analysis is the "dynamic" characterization of nodes' behavior, more realistic than the "static" assumptions the previous cases rely on. Since the Kademlia-like model is a little more complicated, an entire section (3.4) is reserved for developing an exhaustive analysis of it.

3.3.1 The adversary model

The model we've adopted is the *oblivious adversary model.* The oblivious adversary knows the rules according to which the caching algorithm works, but does not know the *randomized results* of the algorithm; that is, the adversary is totally unaware of any network's cache state (i.e., stored values) at a specific moment. It can issue only *one batch* of $(N-\mu)/t$ requests (N is the size of the network, while t represents a constant integer value such that $t \geq 1$): it chooses the $(N-\mu)/t$ *different* nodes belonging to the network (but not to M) that will issue a query for a popular key, and the popular key chosen by each of those nodes as well.

Note that this model lets us remark, once more, that no assumption regarding the employed caching strategy is made: in fact, knowing the values stored by a certain node's cache at a specific time is not necessary.

3.3.2 Routing table with 1-buckets

In this simplest case, as we've said before, each routing table's bucket has a size of 1, storing a reference to only one neighbor. Each of the $(N-\mu)/t$ nodes by means of which we model the oblivious adversary chooses, in a completely random manner, a popular key k_i and issues, in turn, the corresponding query. We claim that

Theorem 2. *In the 1-buckets network model depicted above, if the hot zone M has a size of $\mu = qcD$ nodes, the expected value for the load experienced by at least one of the nodes belonging to M, $E[L_{m_1}]$, is bounded below as follows:*

$$E[L_{m_1}] \geq \frac{q-1}{tq}\left(\frac{N}{qcD}-1\right)$$

Furthermore, it holds that

$$\Pr\left(|L_{m_1} - E[L_{m_1}]| \geq E[L_{m_1}]\right) \leq \frac{2}{e^{\frac{q-1}{tq}\left(\frac{N}{qcD}-1\right)}}$$

.

Proof. Due to property 1, each query reaches an outer border's node in at most $D-1$ hops. If p_{M_1} represents the probability for one of those queries to reach a node belonging to the hot zone, without being intercepted by any of the intermediate nodes' caches, we have

$$p_{M_1} \geq \frac{\mu - cD}{\mu} = \frac{qcD - cD}{qcD} = \frac{q-1}{q} \tag{3.2}$$

In fact, each visited node's cache stores c values and, since property 3 holds, no popular key is more likely to be stored within an intermediate cache than another. Note how that bound is not very tight: indeed, we're assuming that every query passes through a number of intermediate nodes as great as possible and, furthermore, that the intermediate caches store values that are all different from one another. Although a more precise definition for p_{M_1} would be

$$p_{M_1} \geq \left(1 - \binom{\mu-1}{c-1} \Big/ \binom{\mu}{c}\right)^{D-1}$$

we'll consider the inequality (3.2) in this proof, since it is simpler to handle and, moreover, it's enough for our argumentation to hold. In order to find out the number of queries that are expected to really enter the hot zone M at the end of the process, we can model it as a binomial variable Q with parameters $n = (N-\mu)/t$, $p \geq p_{M_1}$; its expected value is bounded as follows:

$$E[Q] \geq \frac{N-\mu}{t} p_{M_1}$$

Note that we can represent the lookup paths followed by every distinct query as independent, identically distributed (i.i.d.) random variables thanks, in particular, to property 4. Since property 2 holds, all these queries, once they have reached M, end up their search path without ever going out from M: therefore, they all can be taken into account when considering the total load produced on M, L_{M_1}. Then, considering that L_{M_1} is distributed (not necessarily in a fair manner) over μ nodes, it must hold that at least one node belonging to M experiments a load L_{m_1} such that

$$E[L_{m_1}] \geq \frac{L_{M_1}}{\mu} = \frac{N-\mu}{t\mu} p_{M_1} = \frac{N-qcD}{tqcD}\frac{q-1}{q} = \frac{q-1}{tq}\left(\frac{N}{qcD} - 1\right)$$

and our first claim follows. Then, the application of the Chernoff bound (3.1) with $\delta = 1$ gives us

$$\Pr\left(|L_{m_1} - E[L_{m_1}]| \geq E[L_{m_1}]\right) \leq \frac{2}{e^{\frac{q-1}{tq}\left(\frac{N}{qcD}-1\right)}}$$

that corresponds to our second statement. □

Note how L_{m_1} decreases when D increases, since, intuitively, the greater is the number of "visited" caches, the greater is the probability of a cache hit. Moreover, setting $q = 1$, we have that $\mu = cD$ and, as we could expect, $E[L_{m_1}]$ could eventually be equal to 0: indeed, in such a case, and if each query really visited as many nodes as possible, it would be more likely that the requested value is found in (at least) one visited cache.

3.3.3 Routing table with k-buckets

In this subsection, we introduce a very simple extension to the previous model: now, every node is assumed to know more than one contact for every different distance; we do not consider 1-buckets anymore, but k-buckets instead, where k is a parameter, representing a constant integer greater than 1. According to this model, every node that issues or simply forwards a query can select one among k neighbors as the next hop; due to property 1, at most k^{D-1} paths are available, all passing through nodes not belonging to M. The adversary model remains unchanged. Our aim is to show how introducing k different choices at every hop does not improve the caching mechanism's effectiveness: that is, the asymptotic load remains unchanged with respect to the previous model (i.e., the 1-buckets network model).

Theorem 3. *In the k-buckets network model, the expected value $E[L_{m_k}]$ for the load L_{m_k} experienced by at least one node belonging to M is bounded below as in the previous case:*

$$E[L_{m_k}] \geq \frac{q-1}{tq}\left(\frac{N}{qcD} - 1\right)$$

Therefore, $\Pr\left(|L_{m_k} - E[L_{m_k}]| \geq E[L_{m_k}]\right)$ *is bounded in an identical manner too.*

Proof. Since property 3 still holds, each one of the (at most) k^{D-1} different paths a query can cover during a lookup process is a valid and feasible path, whichever popular key represents the target; therefore, as in the previous case, we can not make any particular assumption regarding the values stored within the intermediate nodes' caches. For each one of those paths, the probability of entering M is bounded exactly as p_{M_1}; so, what a requesting node can actually do is choosing one path among k^{D-1} in a purely random fashion, since every path has an equal probability p_{M_1} of entering M. As a consequence, each query's probability p_{M_k} of reaching M is approximately:

$$p_{M_k} \approx \frac{k^{D-1} p_{M_1}}{k^{D-1}} = p_{M_1}$$

Since we are dealing with identical probability distributions, we can adopt the same model as before, namely a binomial distribution. This scheme gives us a lower bound for $E[L_{M_k}]$ equal to:

$$E[L_{m_k}] \geq \frac{N-\mu}{t} p_{M_k}$$

A simple substitution of terms gives us the result we've claimed. Since the first statements of both theorems are identical, the Chernoff bound (3.1) can be applied here as before. $\square$

It follows that, as long as the next hop along a lookup path is chosen randomly (property 4), even with a greater flexibility in neighbors selection, no advantage is gained in terms of cache hits.

3.3.4 Application to common DHTs

In order to evaluate the practical implications of the theoretical bound we've calculated above, we set the value of D equal to $\log N$, since it represents the lookup path's maximum length for the majority of the DHTs implemented so far (see Chapter 2). When this hypotesis holds, the value of μ becomes:

$$\mu = qc \log N$$

The following corollary is a straightforward consequence of the results achieved by Theorem 2 and Theorem 3.

Corollary 1. *In a DHT that guarantees $O(\log N)$ routing hop long lookup paths, if each node has a constant cache size c, the expected load $E[L_m]$ experienced by at least one of the nodes belonging to the hot zone is asymptotically*

$$E[L_m] \in \Omega\left(\frac{N}{\log N}\right)$$

This bound is easily obtained. Since, as we've already said, Theorem 2 is substantially equivalent to Theorem 3, we can use only the former in order to obtain the expected load $E[L_m]$ calculated for a generic DHT applying a simple substitution:

$$E[L_m] \geq \frac{q-1}{tq}\left(\frac{N}{qc \log N} - 1\right) \tag{3.3}$$

In particular, note that when, besides t and q, c is constant too, an asymptotic bound for $E[L_m]$ is:

$$E[L_m] \in \Omega\left(\frac{N}{\log N}\right)$$

Considering once again Theorem 2, we can estimate the accuracy of our result, bounding the probability P_{L_m} that L_m differs from the provided value of $E[L_m]$ by a factor greater than 2 as follows:

$$P_{L_m} \leq \frac{2}{e^{\frac{q-1}{tq}\left(\frac{N}{qc \log n} - 1\right)}}$$

The result summarized in Corollary 1 suggests us two important observations. First, note how this bound is asimptotically *the same* we would have obtained *without* any caching mechanism, when $\Theta(N)$ queries are distributed over $\Theta(\log N)$ nodes. Second, a hot zone of size $\log N$, sufficient in order to guarantee the validity of our bound with a constant cache size, does not represent a very high requirement.

3.4 A specific case: Kademlia

Kademlia [24] is probably the most known (and employed) DHT-based overlay nowadays; indeed, many P2P applications developed during the last few years are based on the *Kad* network, the most famous implementation of the Kademlia protocol: popular examples are P2P file sharing systems as eDonkey [1], eMule [2] and aMule [3]. As we'll see in subsection 3.4.1, the routing algorithm implemented by the Kademlia network offers a particular advantage with respect to the models we've analyzed in the previous section. In fact, Kademlia's routing algorithm can be considered *dynamic*, not static as the previous ones: labelling it as "dynamic", we mean that the algorithm is able to adapt itself based on the specific situation the network is in when a query is issued. For example, a node that has already served a high number of queries and that, consequently, answers with more latency than other, less loaded, nodes when it is contacted, is less likely to be selected as the next hop along a lookup path.

This section aims to show that these additional aspects don't affect the applicability of property 4: that is, even if more and more knowledge about the network's status is gained as the time goes by, still routing and caching can be considered as two independent mechanisms in Kademlia. In order to formally prove this fact, we first describe the Kademlia protocol in details: a deeper knowledge of it constitutes the basis for understanding the theoretical proof that follows.

For providing an empirical evaluation of the effectiveness of our results, some simulations have been performed too: the most significant results we've obtained are reported in Chapter 5.

3.4.1 Main features of the Kademlia network

Kademlia takes the same general approach as other DHTs. Every node is assigned a unique, 160-bit long ID, uniformly distributed in the identifier space; every object stored by the network has a key, and keys are uniformly distributed in the same identifier space as nodeIDs. In other words, every object is seen as a $< K, V >$ ($< Key, Value >$) pair, in the same manner as before.

By means of those IDs, the provided lookup algorithm locates successively "closer" nodes to any desired ID, converging to the lookup target in logarithmic many steps. The notion of "closeness" needs a more accurate definition: Kademlia captures the notion of distance by means of the *XOR metric*; indeed, the distance between two nodes, or between a node and a key, is calculated as the bitwise XOR of the two corresponding IDs.

The XOR metric adopted by Kademlia suggests a binary tree-based sketch of the system: Kademlia effectively treats nodes as leaves in a binary tree, where each node's position is determined by the shortest unique prefix of its ID. In Figure 3.2, the location in such a tree held by a node with unique prefix 1100 is shown.

For any given node, the binary tree is divided into a series of successively lower subtrees that don't contain the node; the highest subtree consists of the half of the binary tree not containing the node; the next subtree consists of the

[1] The original client, eDonkey2000, is no longer maintained; however, the eDonkey network (eD2k) supports different clients, like aMule, eMule, MLDonkey and Shareaza.

[2] emule-project.net

[3] amule.org

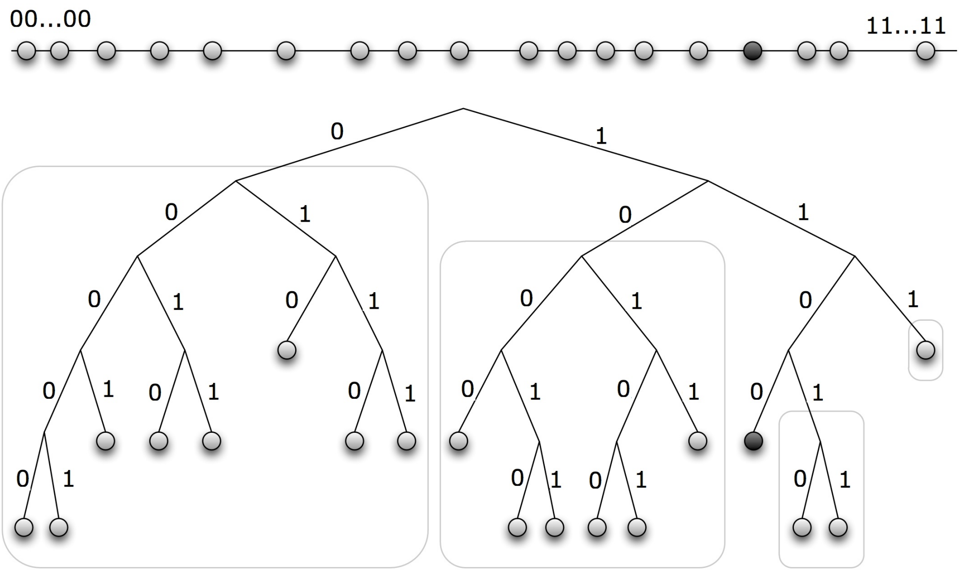

Fig. 3.2: The Kademlia network can be seen as a binary tree, where each leaf corresponds to an active node.

half of the remaining tree not containing the node, and so forth (see again Figure 3.2).

We can consider that, in a fully populated tree of 160-bit IDs, the magnitude of the distance between two IDs is the height of the smallest subtree containing both of them. When a tree is not fully populated, the closest leaf to an ID x is the leaf whose ID shares the longest common prefix of x.

Like Chord's clockwise circle metric, the XOR metric is *unidirectional.* For any given point x and distance $\Delta > 0$, there is exactly one point y such that $d(x, y) = \Delta$. Unidirectionality ensures that all lookups for the same key *converge* along the same path, regardless of the originating node. Like Pastry and unlike Chord, the XOR topology is also symmetric ($d(x, y) = d(y, x), \forall x, y$).

In order to route query messages, every node maintains a routing table with $O(\log N)$ entries, called *k-buckets* (N is, as usually, the size of the network). A k-bucket on level i contains the contact information of up to k nodes that share at least an i-bit prefix with the nodeID of the owner. Kademlia favors long-lived contacts, implementing a *Least Recently Seen (LRS)* policy: a node is placed in a k-bucket only if the bucket is not full or an existing contact is offline.

In the matter of storing, a simple replication scheme is provided: each object is stored on the k closest nodes to the key according to the XOR metric (these nodes are called *replica roots*).

In this context, the storing as well as the query forwarding translate into a *node lookup* operation, performed as follows. The lookup initiator starts by picking α (usually, $\alpha = 3$) nodes from its closest non-empty k-bucket (or, if that bucket has fewer than α entries, it just takes the α closest nodes it knows of); it then sends parallel and asynchronous FIND_NODE RPCs (i.e., Remote Procedure Calls) to the α nodes it has chosen. Once a FIND_NODE RPC has been received, a node looks for the k-bucket nearest to the requested value and returns the requesting node all the values contained in the bucket. In the recursive step, the initiator resends the FIND_NODE to nodes it has learned about from previous RPCs; of the k nodes the initiator has heard of closest to the target, it picks α that it has not yet queried and resends the FIND_NODE RPC to them. Nodes that fail to respond quickly are removed from consideration. If a round of FIND_NODES fails to return a node any closer than the closest already seen, the initiator resends the FIND_NODE to all of the k closest nodes it has not already queried; the lookup terminates when the initiator has queried and gotten responses from the k closest nodes it has seen. A simple exemplification of the lookup process is shown in Figure 3.3: note how, at every hop, the distance between the requesting node and the target is reduced by half with respect to its previous value.

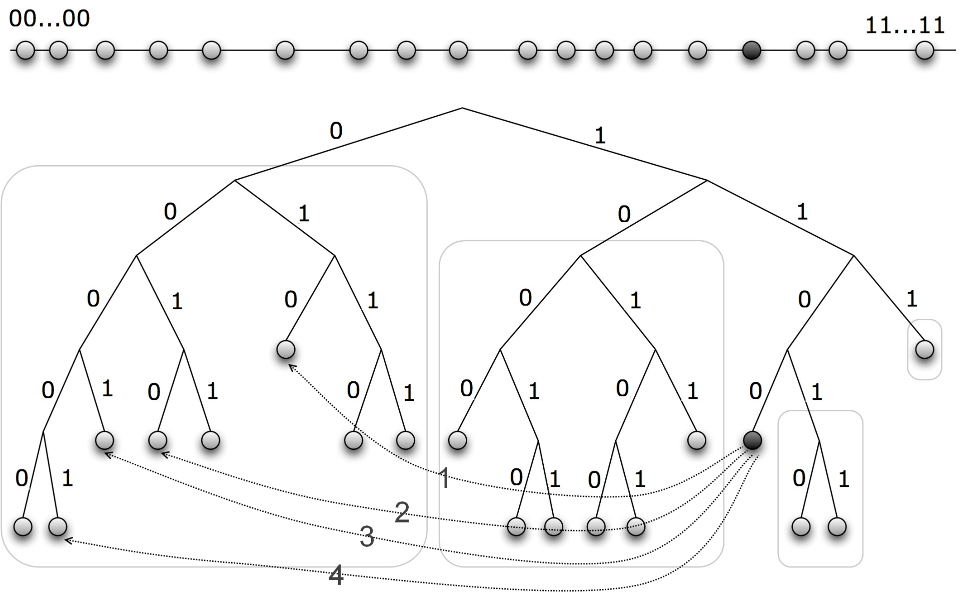

Fig. 3.3: A Kademlia lookup path.

In order to store a $< K, V >$ pair, the peer owning that object locates the k closest nodes to the key and sends them STORE RPCs.

For locating a $< K, V >$ pair, a node first performs a lookup to find the k nodes whose IDs are the closest to the desired keyID; however, value lookups use FIND_VALUE rather than FIND_NODE RPCs, and the procedure halts immediately when any node returns the value.

For caching purposes, once a lookup succeeds, the requesting node stores the $< K, V >$ pair *at the closest node* it observed to the key that did not return the value. According to this mechanism, during times of high popularity for a certain key, the system might end up caching it at many nodes: in order to avoid unuseful replication, a specific expiration time of a $< K, V >$ pair is set in any node's database; this expiration time is exponentially inversely proportional to the number of nodes between the current node and the node whose ID is closest to the keyID.

3.4.2 Probabilistic analysis

Our proof is similar in some regards to those constructed for Theorem 2 and Theorem 3; however, some additional hypoteses are adopted and some adjustments are introduced.

We assume that, as stated in the Kademlia specification, every k-bucket is managed with a LRS policy; in this way, since the network we consider is full (there are N participating nodes), the k neighbors stored within each bucket remain unchanged. Furthermore, the caching mechanism is supposed to adopt a simple *LRU* eviction policy. Due to the "smart" next hop selection implemented, according to which neighbors answering a message more rapidly are preferred (since this means that they are less loaded), we assume that the probability for a node of being contacted during a lookup process is inversely proportional with respect to the number of messages it has already received. In fact, generally, the greater is the number of requests served by a node, the greater is that node's resources consumption (e.g., the available bandwidth), as long as it holds that all requests are issued during a limited time interval (and this is our case).

For simplicity, we consider a hot zone constitued by $qc \log N$ adjacent nodes; without loss of generality, the IDs corresponding to "hot nodes" are assumed to cover the interval starting from 0 in the identifier space, so, according to our convention, the hot zone is placed in the left half of the tree representing the network. The extension of the analysis to a hot zone with different size should be straightforward, since it requires only the modification of some parameters according to the different value of μ.

The adversary model is slightly adapted too; we consider that the adversary can issue *one* query from each one of the $N/2$ nodes belonging to the half of the tree opposite to M (i.e., all the nodes whose ID's most significant bit is equal to 1), choosing an arbitrary popular key as target. The entire process is organized by means of $N/2qc \log N$ *rounds*: at each round, no more than $qc \log N$ nodes issue a query for a popular key and the requested keyIDs are all *different*. Initially, the adversary model we consider is an *adaptive* adversary (not an oblivious one, as in the previous case): it knows (at least in part) the values evicted from each node's cache by the LRU policy, at the end of every round. Theorem 5 summarizes the result we've obtained.

Theorem 4. *In the Kademlia-like network model, the expected load $E[L_{m_a}]$ produced by the adaptive adversary on at least one node belonging to a hot zone constituted by $\Theta(\log N)$ adjacent nodes is asymptotically bounded as follows:*

$$E[L_{m_a}] \in \Omega\left(\frac{N}{\log N}\right)$$

Proof. We assume that, at the beginning of the first round, no node is more loaded than others; furthermore, we consider that every node belonging to the same half of the tree as the hot zone (that is, each node whose ID's most significant bit is equal to 0) stores within its cache the references for c different popular keys, chosen among the μ available popular keys with *uniform probability*; this means that no popular key is more likely to be found within those nodes' caches than another. We assume, furthermore, that this last statement still holds, at the beginning of *each* round i, for the keys that will be requested during round i itself (a justification for this assumption will be provided later in this proof). Thanks to this assumption, the probability p_m that a query message issued during any round enters the hot zone is bounded as above:

$$p_m \geq \frac{q-1}{q} \tag{3.4}$$

Consider the set Q^i, containing all the queries issued in round i:

$$Q^i = \left\{q_0^i, q_1^i, ..., q_{|Q^i|-1}^i\right\}$$

For every round, we define two variables:

- X_j^i: a binary variable, correlated to the occurrence of a cache miss; more precisely:

$$X_j^i = \begin{cases} 1 & \text{if the query } q_j^i \text{ reaches a node belonging to the hot zone (cache miss)} \\ 0 & \text{if the query } q_j^i \text{ is intercepted by a "visited" node's cache (cache hit)} \end{cases}$$

- ν_j^i = popular key requested by the query q_j^i

Due to the uniform distribution of the *requested* popular keys among all the caches, we have that

$$\Pr\left\{X_j^i = 1 | \nu_{j-1}^i, \nu_{j-2}^i ... \nu_0^i\right\} \geq p_m$$

Now, Lemma 1 gives us

$$\Pr\left\{\sum_j X_j^i \leq k\right\} \leq \Pr\left\{B(\log N, p_m) \leq k\right\}$$

In order to complete our proof, we must show that, at the beginning of each round, (3.4) holds, if the appropriate popular keys are requested: in other words, the number of caches storing a requestes popular key must be no greater than the number achieved if every node had chosen the keys to store randomly, with uniform probability.

Due to the employed LRU policy, at the end of round i, $|Q^i|$ *different* values are inserted into some intermediate nodes' caches, and $|Q^i|$ values, not necessarily different, are evicted. The greater is the number of different values evicted, the more balanced is the distribution of the popular keys stored within the nodes' caches: ideally, if the $|Q^i|$ evicted values were all different, the distribution of popular keys among the caches would remain unchanged. If more than one value is evicted from the same node's cache, it's more likely that the values differ from one another (indeed, a cache stores c distinct values); so, we assume that those values are evicted from $|Q^i|$ different nodes' caches. Consider that at round 0 (the first round) all the possible queries are issued, that is,

$$|Q^0| = qc \log N \tag{3.5}$$

The expected number $E[T_0]$ of evicted *different* values at the end of round 0 is equal to:

$$E[T_0] = 1 + \frac{qc \log N - 1}{qc \log N} + ... + \frac{qc}{\log N} \leq \frac{qc \log N}{2}$$

So, we expect than for these $\log N/2$ values (3.4) still holds, while for the remaining $qc \log N/2$ popular keys that assumption is not valid anymore. If, as we've assumed, the adversary exactly knows which popular keys correspond to those $qc \log N/2$ values, it simply does not request any of those during the following round; in this way, at round 1 it issues a total of $qc \log N/2$ different queries, evicting $E[T_1]$ different values:

$$E[T_1] = 1 + \frac{qc \log N - 1}{qc \log N} + ... + \frac{qc \log N/2 + 1}{qc \log N} \leq \frac{3qc \log N}{8} = |Q^2|$$

Similarly,

$$E[T_2] = 1 + \frac{qc \log N - 1}{qc \log N} + ... + \frac{5qc \log N/8 + 1}{qc \log N} \leq \frac{55qc \log N}{128} = |Q^3|$$

Generalizing this reasoning, we obtain that, at round i, $1 < i < N/2qc \log N - 1$, $|Q^i|$ is approximately:

$$\frac{|Q^i|}{qc \log N} \approx \frac{1}{2}\left(1 - \left(\frac{|Q^{i-1}|}{qc \log N}\right)^2\right)$$

In a simpler manner, we can consider only the factor f_i by which $qc \log N$ is multiplied at each round (i.e., the "fraction" of available nodes that the adversary actually uses at round i) and define the following recursive law:

$$f_i qc \log N = |Q^i|$$

$$f_i = \frac{1}{2}\left(1 - (f_{i-1})^2\right)$$

Resolving this recursion, one can see that it does not monotonically decrease, but assumes alternatively greater and smaller values until it eventually becomes stable at a constant value c. Consequently, even assuming s as the minimum (constant) value reached, the number of queries that are expected to enter M at the end of the entire process results:

$$\sum_{i=0}^{N/2qc \log N} |Q^i| \geq s \sum_{i=0}^{N/2qc \log N} qc \log N$$

and, dividing this load by the size of M ($qc \log N$), we obtain an asymptotic bound for the expected per-node load:

$$E[L_{m_a}] \in \Omega\left(\frac{N}{\log N}\right)$$

□

Note that an informal, intuitive analysis would lead us to an identical result even if an *oblivious* adversary model were adopted. Indeed, considering that, at the beginning of the first round, the popular keys are uniformly distributed over the network nodes' caches, the probability, for each of the first $qc \log N$ queries, of entering the hot zone M is bounded as in (3.2):

$$p_M \geq \frac{q-1}{q}$$

For the subsequent rounds, this assumption is not valid anymore; indeed, once the first "batch" of queries has been issued, in $qc \log N$ visited nodes a cached value is evicted (according to the LRU policy), and a new value is inserted (the one the requesting node was looking for). While these new values are all different from one another, two or more evicted values could be the same. Consequently, starting from the second round, there is not a balanced distribution of popular keys over caches: some values maintain an unchanged frequency, some become more frequent, some become rarer. Nodes do not exchange information regarding cached keys with their neighbors, since a node simply *checks* if its cache contains the requested value, whenever it receives a query; therefore, no node holds a complete visual over other nodes' cache state: which values are more frequent, and where those values are placed across the network, is not known by any node. This means that, even if the route followed by subsequent queries is more likely to pass through not yet visited nodes (since they are less loaded), no assumption can be made about the values stored within the nodes that are contacted along the path.

Now, we consider the remaining $N/2qc \log N - 1$ rounds as a unique (random) process; it can not be directly modeled by means of a binomial variable $B(n,p)$ as before, since p is not equal for all the $n = N/2 - qc \log N$ queries. But note that, every time the number of replicas of an object i decreases by one, there exists an object j whose number of replicas increases by the same amount. Since our aim is to estimate the expected number $E[Q]$ of queries reaching M, without considering which popular keys are effectively requested by those queries, $E[Q]$ remains the same as np, since a query for object i is more likely to entering M, while a query for object j is more likely to be intercepted by an intermediate node's cache. Roughly speaking, it seems that discarding more loaded nodes along a lookup path, as long as it is made without caching considerations, does not affect our asymptotical results: after all, this is what our intuition clearly suggests us. Experimental results regarding this problem can be found in Chapter 5.

3.5 Conclusions

The results we've obtained in the previous sections are significant, but they may seem strongly constrained by the assumptions and the semplifications they rely on. Therefore, it's necessary to study in greater detail the actual applicability of our method to more realistic contexts (subsection 3.5.1).

Once we've discussed these fundamental aspects, we'll show how all the four properties listed in section 3.2 are necessary in order to achieve our results; consequently, we'll try to understand which property (or properties) can be removed, in order to provide a better per-node asymptotic experienced load without affecting the DHT's typical efficiency (subsection 3.5.2).

3.5.1 Extending the analysis

In order to be able to develop a theoretical analysis regarding complex phenomenons, it's often necessary to introduce some semplifications, that make the argument less hard to handle. In particular, in our work we've adopted three main hypoteses: all the peers are considered "abstract" and homogenous entities (that is, an identical amount of resources is available at each node), the network is fully populated, the nodes belonging to the hot zone have adjacent IDs. Since the introduction of these aspects could partially affect the validity of our proofs, our aim here is to discuss what limitations, if any, these hypoteses introduce.

Homogeneous network During our analysis, we always consider the *load* experienced by a node as the number of queries it receives (possibly, within a well defined time interval). Therefore, our approach is general and can be immediately adapted in order to take into account a specific aspect (for example, one may consider the loss of performance, the waste of bandwidth, the presence of bottlenecks etc.).

Full network If one considers a not completely populated network (that is, as it often occurs, the number of keys is greater than the number of nodes), it may be possible that one node is responsible for more than one popular key; this would mean that the load experienced by such nodes is even worse than the load we consider in our analysis: therefore, even in this case, the bounds we've found remain valid. Instead, a problem could arise, especially for the Kademlia-like network model, since we do not consider nodes joining or leaving the network, and so we ignore the possibilty of a subsequent update of the routing tables' k-buckets. However, note that this aspect, if taken into account, could make the attack we've described more complicated but, also, bring an advantage for the adversary: indeed, if one node chooses as its new neighbor a hot zone's node, then the caching mechanism is completely "bypassed", since the query reaches immediately the hot zone.

Adjacent IDs This is the strongest and, therefore, the most constraining hypotesis. On the one hand, it's highly unlikely that the hash function mapping keys and nodes into IDs creates such a situation (even if, however, it is not impossible: it depends on what files become popular time after time); but, on the other hand, one could model an ad-hoc, rushing attack that endangers exactly such a set of nodes. Furthermore, our analysis suggests that one should pay attention to some particular situations, in which a node corresponds to an "isolate" ID (that is, there are not nodes having an ID close to it) and, consequently, it may be assigned a large number of values to store, that could eventually become popular.

3.5.2 Necessariety of our assumptions

A starting point for studying a novel caching and routing algorithm is the analysis of the four properties listed in section 3.2 and, in particular, the analysis of their necessity. Once we've achieved this result, it is enough to consider which property we should remove in order to guarantee a more effective caching mechanism.

Therefore, in this subsection we study how, if we remove even one property, the arguments we've exposed can't be accepted anymore.

- *Property 1*: if the Diameter property does not hold, there is not a predefined value bounding a lookup path's maximum length; as a consequence, we are not able to determine how many caches, at most, are visited along a path from the requesting node to the target;

- *Property 2*: without the Distance minimization property, a query issued for a popular key, once it has entered the hot zone, could eventually go out from it one or more times before reaching its target. Therefore, it may occur that, although the query has entered the hot zone at least once, it is then "intercepted" by a node not belonging to it: in this case, that query can not be taken into account when computing the total load L_M experienced by the hot zone.

- *Property 3*: when this property is taken off, it does not hold anymore that every cache "visited" by a query message along a path from the node issuing the request to the node storing the desired value can store any popular key, according to a uniform distribution. In fact, if the path from a requesting node q to the popular node m_i passes through a node n that is not visited when q contacts another popular node, say m_j, n's cache is more likely to store k_i than k_j. At this point, it should be emphasized how the *convergence property* that is typical of most DHTs, according to which search paths with different sources but equal (or close) destinations tend to "converge" along a common path in proximity of the destination, becomes a disadvantage in this context; in effect, one path can serve many different queries and is not able to cache an adequate number of values.

- *Property 4*: clearly, if the routing algorithm is aware of the values stored within the caches of nodes along a lookup path, and chooses the next hops according to this knowledge, the cache hit probability estimation becomes less straightforward and more complicated. In particular, that probability could noticeably increase, until it eventually invalidates the bound we've calculated.

Probably, the first three properties represent the main features a structured P2P network is based on; in effect, they are the most important in order to provide a good search performance in the network. For instance, if Property 1 or Property 2 did not hold, the routing algorithm's convergence would become problematic and the time required for locating a key could excessively increase.

In contrast, the Caching-independent routing property seems to be removable without the risk of affecting the functioning of the system as a whole. So, this could exactly represent the solution we are looking for: the main goal of the following chapter is, in effect, constructing a novel algorithm that couples routing and caching into a unique mechanism, without producing an excessive overhead.

4. CACHE VALUES GUIDED APPROACH

As we've already anticipated, our work aims to construct a novel scheme, whereby the two aspects of routing and caching are strictly interconnected: somehow, we would like the latter to actively "guide" the former. In order to do this, we develop a new P2P network protocol, that presents a well defined geometric interpretation (namely, we adopt an *hypercubic* structure), defines an efficient search algorithm and is able to handle particular phenomenons such as *churn* and *hotspots*.

This chapter exposes our approach in details. In section 4.1, we review the main topological properties of hypercube graphs and briefly consider some significant examples of P2P networks that have already utilized a hypercube-like structure. An intuitive contextualization of the problem we're addressing is provided in section 4.2. Sections 4.3 and 4.4 represent the core of the entire work; recalling that, as we've stated before, a DHT design can be roughly divided into two categories [14], section 4.3 addresses the "*structure level*"(keys storing, neighbors selection, procedures for joining or leaving the network etc.), while section 4.4 deals with the "*routing level*"(routing table organization, length of search paths, next hop choice, recursive or iterative delivery of messages etc.). In section 4.5, expected and worst-case achieved performances are quantitatively analyzed. Finally, in section 4.6, some conclusive considerations are carried out.

4.1 Network geometry

On the basis of the connections it establishes among participating nodes, every P2P overlay resembles, somehow, a specific geometry. For example, the Chord network is tipically seen as a ring, CAN as a torus, Kademlia as a tree; furthermore, other (less known) graphs (among them, butterflies and de Brujin graphs [14]) have been already studied and proposed in networking environments, primarly because they provide interesting and favorable geometric properties. In order to organize our overlay, we've made a precise choice: the *hypercube graph* is the topology we've adopted. Whereas hypercubes have been widely employed in multiprocessor parallel machines (in particular, for models characterized by no shared memory and no global synchronization), hypercube-like P2P networks are less frequent.

In order to better contextualize our overlay, we first review the necessary mathematical background, recalling some important topological properties owned by hypercubes (in particular, we'll focus on those properties we'll take advantage of in our model); then, a few examples of structured P2P based on hypercubic overlays are reported.

4.1.1 The hypercube graph

In this subsection, we briefly review some significant topological properties owned by hypercubes [16, 35]: among them, the "recursive construction property", the presence of parallel paths and other properties that can be particularly useful in networking contexts.

In the following, we'll adopt a "hybrid" terminology: while a formal graph theory generally refers to elements such "vertices", "arcs (edges)" and so on, here we consider those terms as interchangeable with terms like "nodes" or "links". Whereas this could translate into a less formal discussion, we think that such a choice could improve our discussion's clarity.

First of all, in order to develop a rigorous analysis, we provide a formal definition of the hypercube graph.

Definition 3. *A* d-dimensional hypercube *(*d-hypercube*) is an undirected graph consisting of* 2^d *nodes, where each node* n *is labelled by its* d-bits *representation. Two nodes* $n = n_0...n_{d-1}$ *and* $m = m_0...m_{d-1}$ *are connected by an edge (i.e., they are* neighbors*) if they share the same bits but the ith one for some* i, $0 \leq i < d$, *i.e. if their Hamming distance* $H(n, m)$ *is equal to* 1.

An example of 3-hypercube is shown in Figure 4.1.

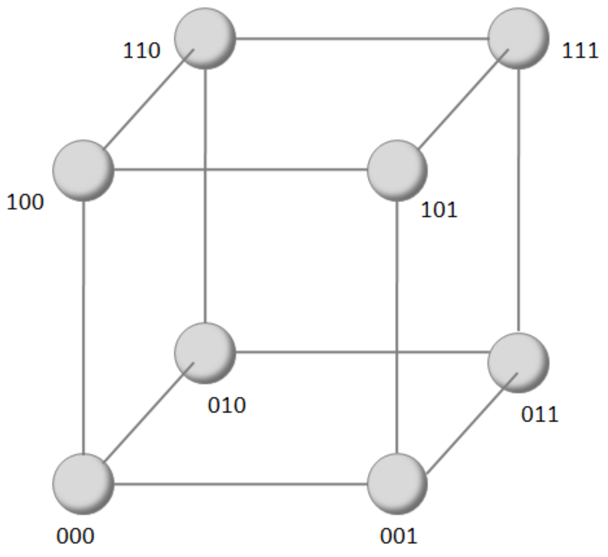

Fig. 4.1: A 3-hypercube: every nodeID is represented by means of 3 bits.

A very important aspect (namely, a direct consequence of the definition) is that every hypercube can be built by means of a *recursive construction*: indeed, a d-dimensional hypercube can be obtained from two identical $(d-1)$-hypercubes whose vertices are numbered likewise, from 0 to 2^{d-1}, by joining every vertex of the first $(d-1)$-hypercube to the vertex of the second having the same number. Then, it suffices to renumber the nodes of the first cube as $0b_i$ and those of the second as $1b_i$, where b_i is the binary string representing the two corresponding nodes of the $(d-1)$-hypercube. Figure 4.2 shows an example of such a construction for a 2-hypercube.

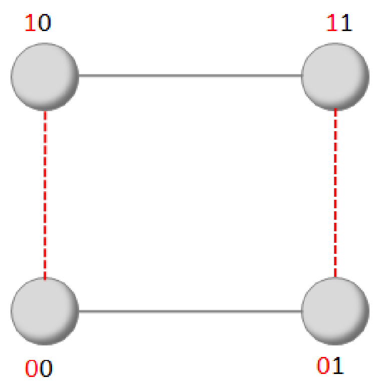

Fig. 4.2: Recursive construction of the 2-hypercube.

Note how, inversely, a d-hypercube can be splitted into two $(d-1)$-hypercubes so that the nodes of the two $(d-1)$-hypercubes are in a one-to-one correspondence; this result is achieved by means of a *tearing process*, that is, just considering the original graph as two subgraphs, the first including all the nodes whose *ith* bit is 0 and the second including those whose *ith* bit is 1 $(0 \leq i < d)$.

Considering this tearing process, it can be easily seen that any two adjacent nodes A and B of a d-hypercube are such that the nodes adjacent to A and those adjacent to B are connected in a one-to-one fashion: since A and B are neighbors, their IDs differ only in one bit, say the *ith* bit; the neighbors of A and those of B are put in a one-to-one correspondence by mapping a node whose label has a 1 in its *ith* position to the one whose label has a 0 in its *ith* position.

As it should be clear, the structure is *symmetric*: indeed, no node incorporates a more prominent position than others, as it is necessary in order to construct a balanced P2P network overlay.

Dimension d represents a very important parameter, that characterizes both the *diameter* and the *degree* of a hypercube. For the sake of clarity, we briefly recall what distance, diameter and degree of a graph mean:

Definition 4. *The* distance *between two nodes in a graph is the number of arcs in a shortest path connecting them. If there is no path connecting two nodes, their distance is defined as infinite.*

Note how, in a hypercube, to reach a node with ID B from a node with ID A, it suffices to cross successively the nodes whose labels are those obtained modifying the bits of A one by one in order to transform A into B. Assuming that A and B differ in i bits (that is, their Hamming distance is $H(A,B) = i$), the length of the path is i; clearly, no path of smaller length can be found between A and B.

Definition 5. *The* diameter *of a graph is the greatest distance between any pair of nodes; in other words, it is the shortest path between most distant nodes.*

Intuitively, the diameter can be seen as the largest number of nodes which must be traversed in order to travel from one node to another without backtrack, detour or loop.

Definition 6. *The* degree *of a graph's node is the number of arcs incident to that node. In a* regular *graph, all degrees are the same, so one usually can speak of the degree of the graph.*

Two important properties held by hypercubes are *low* network diameter and *low* node degree:

Property 1 (Diameter)**.** *The d-hypercube is a* connected *graph of diameter d.*

One can easily see it considering that the distance between two nodes in a hypercube corresponds to the Hamming distance between their IDs. The maximum number of different bits is exactly $d = \log N$, where N represents the size of the network.

Property 2 (Degree)**.** *The d-hypercube is a* regular *graph with degree d.*

Note how the degree is directly correlated to the number of neighbors every node stores within its routing table.

After analyzing these general and abstract properties, we mention two other favorable characteristics owned by hypercubes, particularly significant for the construction of our structured overlay. In particular, we've chosen to emphasize them, among the great number of existing properties, because they concern routing and connectivity aspects.

Specifying that two *paths* are *independent* if they do not share any common node other than the source and the destination, we introduce the "parallel paths" property.

Property 3 (Parallel paths)**.** *Let A, B be any two nodes of a d-hypercube and assume that $H(A, B) < d$. Then, between the nodes A and B, there are $H(A, B)$* parallel *paths of length $H(A, B)$ and d* parallel paths *whose length is at most $H(A, B) + 2$.*

In fact, a path from a node labelled as A to a node labelled as B is obtained by crossing successively the nodes whose labels are obtained by modifying one by one A's bits into B's bits, in order transform A into B. If $H(A, B) = b$, then b independent paths between A and B can be found as follows: path i is obtained by successively correcting bit i, bit $i + 1$, ..., bit $(i + b - 1) \bmod (b)$ among the b different bits between A and B. These b paths are of optimal length $H(n, m)$. In addition, $d - b$ paths of length $H(A, B) + 2$ can be constructed as follows: path j of length $H(A, B) + 2$ is obtained by modifying the first bit j on which A and B agree, then correcting the b different bits according to one of the b possibilities described previously, and finally remodifying bit j. Figure 4.3 exemplifies this mechanism, showing the four parallel paths existing between nodes 0001 and 1011 in a 4-hypercube.

It's important to remark how, in P2P networking environments, the parallel paths property brings two important advantages: first, it enhances fault tolerance

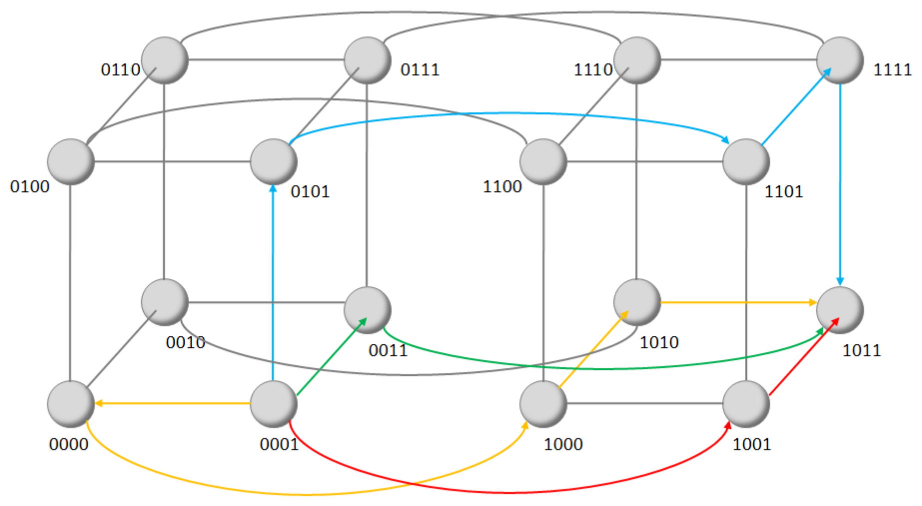

Fig. 4.3: Parallel paths between a pair of nodes in a 4-hypercube.

(if a faulty peer is encountered along a path, then various alternative routes can be easily found); second, it eases a full utilization of every node's available bandwidth: in fact, a node holds $d = \log N$ links that connect it to the rest of the network, and a message can theoretically reach it passing through any of those links with an equal probability.

Finally, a property strongly related to *robustness* (that is, how well a network can sustain node failures) is the following:

Property 4 (Connectivity)**.** *In the d-hypercube, the minimum number of nodes (or links) whose removal results in a disconnected network is equal to d.*

This property provides us an useful bound for the number of (almost) simultaneous failures an hypercubic-like network can sustain without its whole working being affected.

4.1.2 Hypercubic P2P networks

The introduction of our hypercubic overlay is not an absolute novelty in P2P environments: a few significant examples are available in literature [37, 3, 2]. This subsection briefly describes those schemes.

HyperCuP HyperCuP (Hypercube P2P) [37] can be seen as a "hybrid" approach, that adopts a structured scheme in order to make the flooding lookup mechanism, typical of unstructured networks, more efficient. The peers are organized into a hypercube (or, more general, Cayley graph [1]) topology; the proposed lookup algorithm takes advantage of this structure and implements a broadcast

[1] A Cayley graph encodes the structure of a discrete group: if G is a group and $S \subseteq G$ a set of group elements not including the identity element, the Cayley graph associated with (G, S) is the directed graph having one vertex associated with each group element and directed edges (g, h) if $gh^{-1} \in S$.

forwarding scheme, which guarantees that the set of traversed nodes strictly increases during the process until all nodes are reached. In particular, the broadcast algorithm guarantees that every node is contacted exactly once. In this way, $N-1$ messages are sufficient in order to reach all the peers belonging to the network, and the last (that is, most distant from the source) nodes are reached after $\log N$ forwarding steps. HyperCuP adopts a well defined overlay, so it can be considered a structured network; nevertheless, differently from our approach, it still adopts unstructured-like searching techniques: even if the performance provided by "blind" flooding methods is enhanced by this scheme, load balancing problems are not taken into account. Indeed, a node could easily become overloaded, since, according to the lookup mechanism mentioned above, it receives *all* the requests sent by its peers, even if it is not the target nor a node located along the shortest lookup paths.

PeerCube A more typical DHT-based system is PeerCube [3]. This architecture brings together DHT schemes with other distributed systems' concepts (for example, *clustering*), offering two main characteristics: nodes sharing a common prefix gather together into clusters, and clusters self-organize into a hypercubic topology. Peers within each cluster are organized by means of a two-layer hierarchy, and classified into two categories (core members and spares), among which only one (the former) is actively involved in P2P networking operations and communications. Locating an object identified through a key k consists in walking the overlay by correcting one by one and from left to right the bits of the requesting node's identifier to match k. The request is propagated until finding either a peer of a cluster labelled with a prefix of k, or no cluster closer to k than the current one.

In order to achieve fault tolerance, a width path approach is employed: a request is forwarded not only to the target cluster, but to a sufficiently big number of clusters, chosen among the closest to the target, so that at least one correct (and on-line) peer can receive it. In PeerCube, the hypercubic structure is introduced mainly in order to provide fault tolerance, allowing operations to be successfully handled despite the corruption of some nodes, and to adequately handle churn. While this work shares some aims with ours (namely, the attempt to prevent malicious attacks coordinated for affecting the system's functioning) and also some routing algorithm characteristics, we adopt a simpler overlay, without hierarchical organization, and introduce a caching mechanism, not implemented in PeerCube.

Hypercubic Hash Table The model depicted in [2] is different from the previous two, since it does not provide a well defined network structure. The hypercube topology, here, is introduced mostly in order to solve a load balancing problem, that is represented as a particular stochastic process: a DHT is considered, in which keys are partitioned across a set of processors, and participating peers are assumed to grow dynamically, starting from a single node. It is requested that, if at some stage there are n nodes, the number of forwarded queries during a lookup is $\log N + O(1)$, the number of values maintained within each node's routing table is $\log N + O(1)$ and, finally, the worst ratio between any pair of processors' loads is $O(1)$ with high probability (that is, load balancing among nodes is achieved). The Hypercubic Hash Table guarantees these results thanks to a complex paradigm based on the balanced allocation mathematical theory: every node, when joining the network, must follow that well defined paradigm in order to keep the entire network balanced. Whereas this scheme does not represent a complete P2P specification, since it does not address crucial problems like leaving or failing nodes and does not elaborate a rigorous lookup algorithm, it is useful to remark, once more, the advantages offered by hypercube graphs in terms of balancing and efficiency.

Our approach addresses a different kind of problem with respect to the overlays just described, so, as we shall se, it will introduce new and original aspects; to our knowledge, our hypercubic-like P2P network is the only one that addresses the caching-routing cohesion problem in order to avoid flash crowds and hotspots. However, our structure maintains all the favorable characteristics achieved by the works exposed above, especially in terms of robustness and load balancing.

4.2 A first intuitive approach

Before explaining in details our scheme, we present in this section a sketch of a slightly different, interesting approach. This should not be considered a totally irrelevant discussion: indeed, it represents a kind of "first step" in order to bone up on the study of our problem, identifying its manifold and various challenging aspects. In effect, this system represents the first "prototype" we studied and developed in our work's initial phase; it reveals some fundamental ideas that will be kept unchanged by our definitive scheme (exposed in sections 4.3 and 4.4), and, at the same time, presents some significant weak points, that compromise the efficiency provided in some particular networking configurations. It is exactly on the basis of the limitations shown by this technique that we can develop a new, more feasible strategy.

First of all, subsection 4.2.1 emphasizes the difference, already mentioned in Chapter 2, between *destination load* and *routing load*: this distinction is a crucial point, that directly justifies the caching mechanism presented in subsection 4.2.2. Subsections 4.2.3 and 4.2.4 quantitatively evaluate the model. Finally, subsection 4.2.5 points out the main limitations suffered by this scheme.

4.2.1 Request load and routing load

As we've seen in Chapter 2, the "query" load experienced by a node can be generally classified into two distinct categories: *request load* and *routing load.*

Indeed, every query received by a node either requests an object stored by the node itself, so no subsequent forwarding is necessary, or requests an object stored in another location in the network, so it must be forwarded to an appropriate neighbor. In the first case, the query increases the request load; in the second case, the query is included in the routing load.

As we've seen in the previous chapter, it is the *second* type of load, produced by queries simply forwarded towards other contacts, that plays a major role in producing a high per-node overload. In effect, even if, in our model, the adversary chose target IDs as different as possible, the total load experienced by the hot zone's nodes would remain the same: we don't concern with which particular objects are queried, since it's enough to know that they belong to the hot zone.

This characteristic, when combined with the typical "*rigidity*" owned by DHT, leads to quite negative effects. The term "rigidity" refers, in particular, to some properties that we have already explained in details and here recall once more, as the prefix-guided property and the independence between routing and caching.

Note how a direct consequence of the prefix-guided property is the subsequential *convergency*, that characterizes most DHT routing algorithms. This aspect is normally considered a very useful feature. According to the convergency property, queries sent by different nodes (even far enough from one another) tend to "be reunited" into a very similar path near the target: while the first hops span over very large distances, the last hops represent small improvements, simple "refinements" that move within a restricted set of nodes, adjusting carefully the least significant bits of the sender's ID. But let us consider the other side of the coin. The prefix-guided property causes a lack of flexibility in the routing algorithm, that substantially prevents an interaction with the caching scheme adopted by the network. Routing paths follow well defined rules in order to reach their target, and these rules are not changeable depending on the values stored at nodes encountered along the lookup path. Moreover, the ordered adjustment of ID's bits "from-left-to-right" produces an important effect, that we've already analyzed in the previous chapter. Lookup paths originating from the same requesting node and directed toward IDs that are different, but located in positions that are close in the identifier space, can share a large number of intermediate hops visited during the first part of the path, while they differentiate in proximity of the target. The nodes located along that "common" path receive a lot of queries requesting *different* objects, so, in some cases, they can not make caching effective. Such an intermediary node becomes a kind of "bottleneck", wasting bandwidth and computing resources only for serving requests directed to other, different targets, and can't efficiently act as a "shield" for the nodes that are reached through it by means of caching.

On the basis of these considerations, an obvious requirement for our novel approach is, first of all, a greater *flexibility*.

It is this need of flexibility that has suggested us the introduction of an hypercube-like overlay. In such a network, the IDs nodes and keys are hashed into correspond exactly to the verteces of the hypercube graph. Therefore, the definition of a node's neighbor (that is, contact stored within a node's routing table) is easily derived.

Definition 7. *Two (distinct) nodes are said to be* neighbors *if there exists a direct link between them in the hypercube overlay.*

The hypercubic overlay lets us adopt a *randomized* routing algorithm, able to adapt itself in order to pass through the intermediate nodes that are more likely to store the requested value within their cache. In particular, if node A issues a query directed to node B and $H(A, B) = k$, A can choose in a totally random manner one among the $k!$ minimal length paths obtained by flipping the bits that differ from A's ID to B's. More precisely, A chooses a first "hop", say node C, among k available choices; C forward the query selecting one among $k - 1$ neighbors, and so on: the implemented algorithm is *recursive.*

The routing-caching interconnection is achieved by means of two particular concepts, explained in details in the following subsection.

4.2.2 The forwarding value and adaptive routing concepts

The previous subsection has underlined how an important factor in producing overloaded nodes is the routing load suffered, that can be much more harmful than the destination load.

Therefore, we would like a node to be able to *block* incoming queries that cause an excessive routing load, trying to direct them toward different, possibly more adequate and less loaded, destinations.

Before starting a brief description, we provide some preliminary concepts.

Definition 8. *The* inner border I *is the set of all the nodes* i_j *that belong to the hot zone* M *and have at least one neighbor not belonging to* M.

Definition 9. *A node (either belonging to* M *or not) is called a* inner node *if each one of its neighbors belongs to* M.

In the following, we'll assume to deal with a full network: in such a way, it will be easier to understand the most important principles at the basis of our reasoning. Furthermore, in this subsection, we limit our study to situations where inner nodes are not present and we consider that, for each node n belonging to the hot zone M, the number of paths directed to n which have a node belonging to M as next-to-last hop are less numerous than the number of paths having a node not belonging to M as next-to-last hop. More precisely, it is assumed that

$$\forall n \in M, \sum_{i=1}^{m} p_{n_i} \geq \sum_{j=1}^{n} q_{n_j}$$

where p_{n_i} and q_{n_j} are n's neighbors, $p_{n_i} \in M$, $q_{n_j} \notin M$, $m + n = \log N$. In subsection 4.2.3, we'll see how this assumption is not so restrictive: in effect, it has a high probability to hold. In subsection 4.2.5 we'll remove such hypoteses, extending the analysis to more complicated (and rare) networking configurations where, unfortunately, this scheme is not robust enough.

The key idea introduced in order to avoid an excessive routing load is the *forwarding value.* The forwarding value is a kind of "flag", stored within each node's routing table entry. It can be thought of as a simple bit, that is set at 0 ("*not forwarding*", or "*blocking*", state) or at 1 ("*forwarding*" *state*). Figuratively, the forwarding value acts as the "semaphore" utilized by operating system environments: it controls access by several nodes (namely, the neighbors) to a common resource (namely, the node itself). If a node is in "blocking" state, no query must be forwarded toward it; in contrast, if it is in "forwarding" state, network traffic can pass through it as usual. Whenever the node's state changes, the corresponding flag within each one of the neighbors' routing tables must be updated accordingly. The whole mechanism becomes clearer when we analyze in greater detail how every node handles this forwarding value when it *updates* or *consults* its routing table:

Setting Every peer cyclically checks if the load it is experiencing goes over a predefined threshold or not (the load can be calculated in terms of received queries within a certain time interval, consumpted bandwidth etc.); when that threshold is reached, the node becomes a "hot" node. Therefore, it stores the object it is responsible for (recall that, in our model, each node is responsible for -at most- one object) within each neighbor's cache, and, in each neighbor's routing table, sets the forwarding value corresponding to itself to 0. This flag can be associated with a Time To Live (TTL) value, so that the bit is flipped to 1 when the TTL expires and "renew" request is not received. Observe that it could be enough that every node takes into account, in order to calculate its load, only the received queries requesting the value its own, not the queries to forward. In fact, it is likely that a node storing a popular value gets overloaded *before* any of its neighbors, considering that the adopted routing algorithm makes use of randomness. The load of forwarding these requests should be shared out, if not fairly among all the $\log N$ neighbors, at least among a few of them.

Consulting Whenever a node receives a query to forward, it selects, in a totally random fashion, a neighbor from its routing table; in order to be definitively chosen as the "next hop", this neighbor must satisfy two requirements: it must be closer to the target than the current node and it must be in forwarding state. If, after consulting all the entries, no such a node is found, the query is blocked. An error message is then sent back to the requesting node.

It is clear how the main goal of the entire mechanism is to *block* a lookup path when it contacts a popular node that is not the desired target: ideally, we would like a great enough number of replicas of every popular key to be stored within outer border's nodes caches; queries coming from nodes not belonging to the hot zone should be directed toward the appropriate outer border's nodes, so that they can be intercepted and find the requested value without entering the hot zone. The net result is that the routing load over popular nodes is completely removed.

But, what is the price to pay in terms of overhead produced by this mechanism? And, moreover, is this scheme robust against *any* type of attack aiming to overwhelm with an untenable amount of traffic one or more nodes? The analysis developed in the following subsections answers exactly these questions.

4.2.3 Routing and caching efficiency

In order to evaluate the scheme's efficiency, we'll briefly discuss the following points:

- since some queries are simply blocked, compelling the requesting node to resend its query, how can the bandwidth/latency tradeoff be handled? We can't unacceptably increase latency values while trying to avoid some node wasting its bandwidth;
- which among the four properties, tipically owned by DHTs (see section 3.2), are here preserved?

In order to answer the first question, we estimate the per-blocked-node number of neighbors belonging to the outer border.

Expected number of outer border's neighbors First of all, we "reverse" the question, trying to calculate the expected number of neighbors belonging to the hot zone M, for a generic node $m \in M$.

Consider that M contains m plus other $\mu - 1$ different peers, that are chosen in a totally random manner among $N-1$ available nodes. We model the problem by means of a hypergeometric random variable with parameters $\mu-1$, $N-1$ and $\log N$: if V represents the number of m's neighbors belonging to M, we obtain:

$$\Pr\{V = i\} = \frac{\binom{\log N}{i}\binom{N-1-\log N}{\mu-i}}{\binom{N-1}{\mu-1}}, i = 0, 1, ..., \mu - 1$$

The expected value is equal to:

$$E[V] = \frac{\log N(\mu - 1)}{N - 1} \approx \frac{\log N\mu}{N}$$

More concretely, we can set μ at a well defined value (for instance, $\mu = \sqrt{N}$ or $\mu = \log N$) and derivate the corresponding $E[V]$:

- $\mu = \sqrt{N}$:

  $$E[V] = \frac{\log N\sqrt{N}}{N}$$

  Approximating the value just found by means of a binomial variable, where $np \approx \frac{\log N\sqrt{N}}{N} = \frac{\log N}{\sqrt{N}}$, we bound the probability of having more than $np \approx \frac{\log N\sqrt{N}}{N} = \frac{\log N}{\sqrt{N}}$ popular neighbors:

  $$\Pr\left\{V > \frac{\log N}{2}\right\} < \left(\frac{e^{\sqrt{N}/2}}{(\sqrt{N}/2)^{\sqrt{N}/2}}\right)^{\frac{\log N}{\sqrt{N}}} = \left(\frac{2e}{\sqrt{N}}\right)^{\frac{\log N}{2}}$$

  As a consequence, the estimated probability p_o of reaching a popular node arriving from a not popular node is:

  $$p_o > 1 - \frac{1}{\sqrt{N}}$$

- $\mu = \log N$:

$$E[V] = \frac{\log N \log N}{N}$$

  Once again, we approximate using a binomial variable with $np \approx \frac{log^2 N}{N}$. The probability of having more than $\frac{\log N}{2}$ popular neighbors is:

$$\Pr\left\{V > \frac{\log N}{2}\right\} < \left(\frac{e^{N/2\log N}}{(N/2\log N)^{N/2\log N}}\right)^{\frac{\log^2 N}{N}} = \left(\frac{2e\log N}{N}\right)^{\frac{\log N}{2}}$$

  The probability p_o, defined as in the previous case, results:

$$p_o > 1 - \frac{\log N}{N}$$

Routing properties Let us discuss which properties remain unchanged, and which are crippled:

- Diameter property: since it's possible that a sender has to resend, many times, the same query before finding a "correct path" toward the destination node, the number of necessary hops can be greater than $\log N$; however, since the probability of reaching the requested popular key coming from the outside of the hot zone is equal or greater than the probability of coming from the inside, the expected number of sendings is equal or less than 2; therefore, the search path length keeps an upper bound of $O(\log N)$ hops;

- Distance minimization property: we are still using a *greedy* algorithm, such that every hop decreases the (Hamming) distance between the current node and the target by 1: this property maintains its validity;

- Prefix-guided path property: since the routing algorithm adopted is randomized, no assumption can be made regarding the order according to which bits within each ID are adjusted. Then, this property does not hold in general;

- Caching-independent routing property: clearly, this property is removed: indeed, a query is not forwarded to a node, if that node surely does not cache the requested value.

As we can see, property 1 and property 2 still hold: this fact is sufficient in order to guarantee a good routing performance over the whole network.

Furthermore, a slight improvement can be introduced. In fact, sometimes it could happen that the first issued query is not able to find the appropriate outer border's node and, therefore, it is simply blocked: in this case, the requesting node must resend its query, so that the latency generally increases. Note how performance could be enhanced using *random walks*. Since the expectation is that k walkers after T steps reach roughly the same number of nodes as 1 walker after kT steps, we can increase the number of queries sent "in parallel" by each requesting node, in order to diminish the latency due to "blocking" situations.

Multiple-walker random walks require a mechanism to terminate the walks, as TTL or checking. With the former, each random walk terminates after a certain number of hops; with the latter, a walker periodically checks with the original requester before walking to the next node.

4.2.4 Simple comparison against Kademlia

This subsection presents a brief comparison between the approach just defined and Kademlia.

As stated in the previous subsections, a certain "*flexibility*" is necessary in order to construct a robust and efficient P2P overlay. Well, the "hypercubic" scheme offers flexibility in routing paths selection, not so much in neighbors selection; Kademlia seems to present an antithetical situation: a node can determine its neighbors with more "liberty of choice", but, in general, routing must obey the "prefix-guided" rule.

In order to make a direct comparison, let us consider how both algorithms work in a specific case. The model we adopt presents a hot zone of size $\sqrt{N}$, where hot zone's nodeIDs belong to interval $\left[0, \sqrt{N}-1\right]$. In this way, all the nodeIDs belonging to the hot zone share a common prefix of length $\log N/2$. As we'll immediately show, this is an *optimal configuration* for the "hypercubic" model: there always exists a way in which the requested value can be found (in other words, no additional queries must be issued, after the first).

Figure 4.4 shows the configuration we're describing, introducing a particular representation of a hypercube (in the following, we'll refer to it as the "*multilevel representation*"): for a network containing (at most) N nodes, such a representation shows $\log N$ levels (conventionally, the ID without any bit equal to 1 belongs to level 0, IDs with 1 bit equal to 1 belong to level 1 and so on). This representation is useful in order to show how the mechanism works for a hot zone M of size $\sqrt{N}$ (in particular, when the hot zone contains the IDs from 0 to $\sqrt{N}-1$). Note that, in this case, for each of the "hot" nodes, half of the neighbors belongs to M, half does not, so the probability of arriving "from the outside" equals to 1/2.

Gray nodes are in blocking state: red lines represent "not available" links (they correspond to links connecting a node in blocking state to its neighbors). Since these links are bidirectional, only those directed toward blocked nodes should be considered not "viable"; however, a node in blocking state does not send any message, so the link can be seen exactly as "failed", in both directions. Consider a query for ID 0000 issued by node 1111. The first hop, 0111, is randomly chosen; 0111 can't forward the query to 0011, so it selects its neighbor 0100. 0100 is an outer border's node: it can not forward the query toward the actual target, but it must store the corresponding value within its cache. Note how, according to this approach, a query "moves" along the outer border, without entering the hot zone, until it finally finds the value it was looking for in a node's cache (cache hit).

It can be shown that *every* routing path starting from the outside of the hot zone does not enter a "dead-end street".

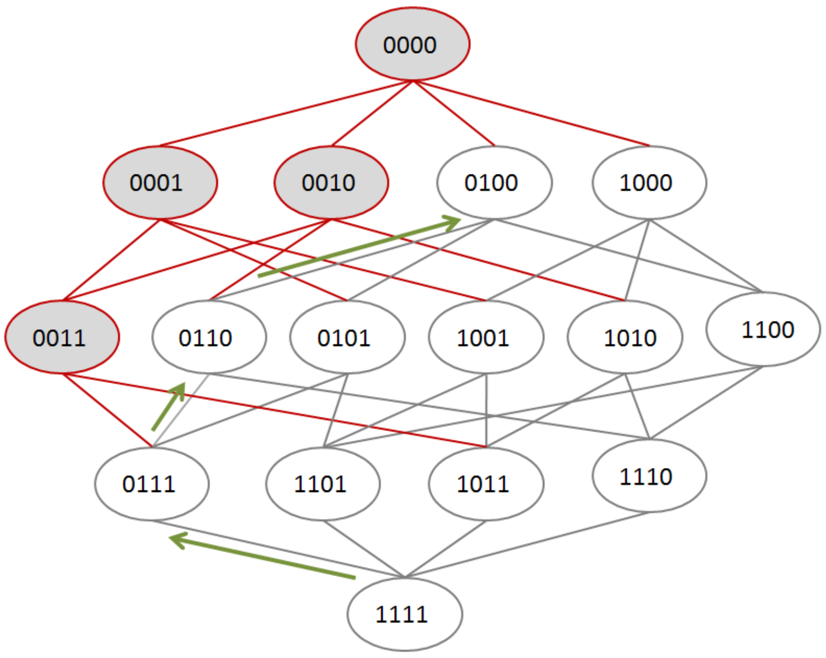

Fig. 4.4: Lookup path (green arrows) from node 1111 to node 0000: the target node is in blocking state, so the key it is responsible for must be cached at the outer border' nodes 0100 and 1000.

Theorem 5. *In the "hypercubic" model, with a "optimal configuration" as that depicted above, every popular key can be found without the need of "resending" multiple times the corresponding query.*

Proof. Our proof is done by contradiction: suppose that such a dead-end street exists, and that its last hop is a node n. n must belong to the outer border and must not have any neighbor to correctly forward the query to. Since n belongs to the outer border, its ID's $\log N/2$ bit long prefix must differ from hot zone nodes' $\log N/2$ bit long prefix in one and only one bit. If the last $\log N/2$ bits of n's ID correspond to those of the requested ID, n is a neighbor of the target, so it must have the desired value within its cache; otherwise, it can "adjust" bit by bit its $\log N/2$ bit long suffix, never entering the hot zone (the prefix remains unchanged), until it finds the appropriate neighbor. □

Now, let us consider the same situation occurring in a Kademlia-like network. As before, hot nodes share an identical prefix, say $t_0, t_1, ..., t_{\frac{\log N}{2}-1}$. Recall the Kademlia tree: if we gradually *flip* the last bit of this prefix, decreasing at each step the prefix length by one bit (that is, we consider prefixes $t_0, t_1, ..., \bar{t}_{\frac{\log N}{2}-1}$, $t_0, t_1, ..., \bar{t}_{\frac{\log N}{2}-2}$ and so on), we obtain adjacent tree zones, each containing twice the number of nodes contained in the previous one. If the requested node corresponds to ID t ($t = t_0, t_1, ..., t_{\log N-1}$), each zone has half the probability of storing t in its routing table with respect to the previous zone.

First of all, note that in Kademlia links are *unidirectional*. So, if a node wants to set the "forwarding value" corresponding to it in other nodes' routing tables, it can not know, in principle, which nodes it should contact. However, even considering that this update can be accomplished without many problems, there is not a bounded number C of neighbors to contact. Taking into account the observation escposed above (regarding the Kademlia tree structure), an expected value can be estimated as

$$E[C] = 2 + \sum_{i=1}^{\log N/2-2} = \frac{\log N}{2}$$

that is, the length of the prefix characterizing the hot zone. Note how this value is not constant, but depends on the size μ of the hot zone: in particular, $E[C]$ increases if μ decreases ($E[C] = \log N - \mu$).

Now, consider a node q requesting key t. Suppose that q belongs to the half of the tree opposite with respect to t (that is, $q_0 = \bar{t}_0$). In order to get the requested value, q must reach t itself or another node, neighbor of t, caching the desired key. Repeating an argumentation similar to that presented for $E[C]$, the expected number of such neighbors, located in the same half of the tree as t, is

$$E[V] = \frac{\log N}{2}$$

Therefore, the probability p_t that a node belonging to such a zone has t as neighbor is:

$$p_t \approx \frac{\log N}{2}\frac{2}{N} = \frac{\log N}{N}$$

Since the maximum length of a search path is $\log N$, the probability p_b for a query to be blocked, without finding the requested value, is

$$p_b \geq \left(1 - \frac{\log N}{N}\right)^{\log N}$$

.

4.2.5 Introduction of inner nodes

In worse configurations, one of more *inner nodes* can exist. Since we want our principle (that is, queries must be "blocked" at the outer border, but they must find the requested value with $O(\log N)$ hop long lookup paths) to hold even in this case, a more complex strategy is necessary. Without an appropriate caching mechanism, objects stored by inner nodes would become unavailable: references to these objects must be somehow "pulled" towards the outer border and cached at some outer border's nodes, so that query messages have a chance of finding them.

Note that one or more entire levels of inner nodes can be created. Furthermore, a not-overloaded node can become a inner node: for instance, this happens when a whole level in the multilevel representation enters the "not forwarding" state. An example is depicted in Figure 4.5: nodes belonging to level 0 and 1 become unreachable, even if they still work correctly.

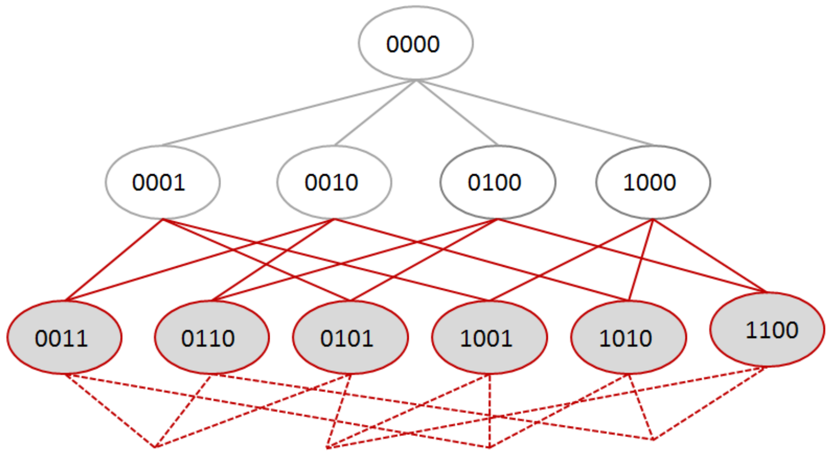

Fig. 4.5: Nodes belonging to level 2 enter the "not forwarding" state; nodes belonging to level 0 and level 1 become inner nodes.

Therefore, it is necessary to introduce a mechanism in order to handle such situations: a natural way for addressing the problem is the utilization of a "*broad-cast*" mechanism.

According to this mechanism, when *at least half* of its neighbors become "blocked", a node, even if not overloaded, enters the "not forwarding" state itself, and "floods" all these neighbors with the key(s) it is responsible for. Note that a node is always aware of the number of its "not reachable" neighbors, since it stores this information within its routing table (recall the "forwarding value" bit).

It is clear that, when there exists a configuration similar to that shown in Figure 4.5, a simple flooding of the key does not produce an acceptable result if "blocked" nodes do not perform any action at all. For instance, the key 0000 would remain unreachable since it can't reach the outer border. Consequently, in some cases, it is necessary that keys received by a "not-forwarding" node are forwarded as well, down along the "leveled" graph, following some path until an outer border's node is finally reached. In this way, we are sure that, whenever a node becomes not reachable (or not *efficiently* reachable, that is, not reachable within a constant number of trials) from other peers (because it is in "not-forwarding" state or because more than half of its neighbors are in such a state), the keys it stores are available in a large enough portion of reachable nodes.

After all, it's reasonable that a node "blocks" only those routes, directed toward itself (remember the *bidirectionality* of links), that carry an excessive traffic; other links should remain available. Considering once more Figure 4.5, a node belonging to level 1 blocks the links coming from level 2, since it is from that direction that traffic arrives; links connecting level 1 to level 0 are kept available. In this way, a node in blocked state simply does not cache the values it receives from other "blocked" neighbors, but forwards them instead; a node in normal state tries to cache all the received values, according to a LRU policy.

Unfortunately, some problems could arise just the same. If we have a large number of adjacent nodes all in blocked state (located within a compact "hot zone"), it is possible that the value owned by an inner node is replicated too many times in the outer border's nodes.

An example is shown in Figure 4.6; suppose that every node, once it knows that it's "unreachable" (that is, at least half of its links are closed or connect it to a blocked neighbor), sends the value it is responsible for along the "broken" links; the neighbors, in turn, forward the value they've just received downward along the leveled graph. If we assume that an outer border's node simply stores the values it receives, without any subsequent forwarding, the load experienced by such nodes could become excessive. Indeed, consider the worst case of "totally closed" hot zone, as that depicted in Figure 4.6: every node belonging to the first two levels forwards (green and blue arrows) its value downward. Whereas the figure shows the mechanism only for two nodes (0000 and 0100), all the nodes belonging to level 0 and level 1 forward their value.

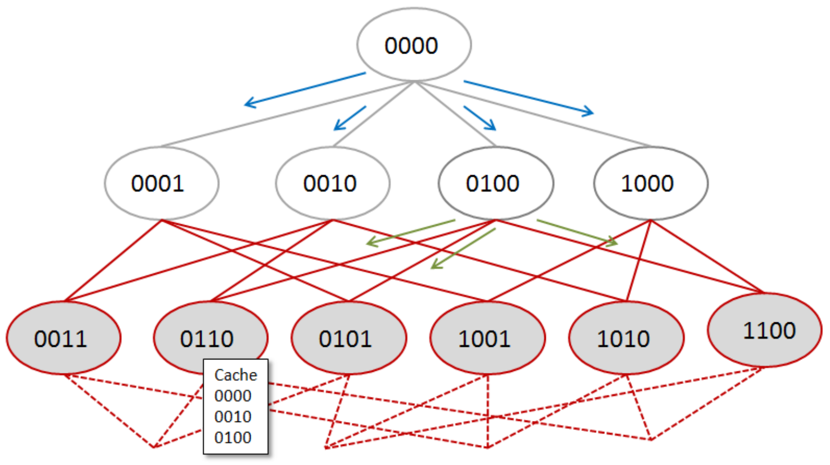

Fig. 4.6: Broadcast mechanism in a congestioned hot zone.

Note that, if we try to calculate the number of nodes belonging to each level, we obtain:

- Level 0 : 1 node; $l_0 = 1$
- Level 1: $l_1 = l_0 \log N$, that is $\log N$ nodes;
- Level 2: $l_2 = l_1(\log N - 1)/2$, that is $\frac{\log N(\log N-1)}{2}$ nodes;
- Level 3: $l_3 = l_2(\log N - 2)/3$;
- Level i: $l_i = l_{i-1}(\log N - i + 1)/i$

Given specific values of μ, the number of items an outer border's node could be asked to store is bounded in the following manner:

- $\mu = \log N$**:** this is the simplest case. If M has a size of $\log N$ nodes, clearly at most 1 node will be an inner node i. Therefore, also if we replicate the value stored at i at every outer border's node, the load experienced by each of them will be $O(1)$.

- $\mu = \sqrt{N}$**:** the amount of inner nodes is larger than in the previous case. Trying to estimate at which level x we reach a total of $\sqrt{N}$ nodes, we sum up the number of nodes belonging to each level:

$$1 + \log N + \frac{\log N(\log N - 1)}{2} + ... \leq \sum_{i=1}^{x} \frac{\log^i N}{i}$$

We are looking for a value x such that:

$$\sum_{i=1}^{x} \frac{\log^i N}{i} \approx \sqrt{N}$$

So, we obtain that

$$x \approx \frac{\log N}{\log \log N}$$

In other words, a hot zone of $\sqrt{N}$ nodes with the greatest amount of inner nodes is represented through a "leveled graph" of approximately $\log N/\log\log N$ levels. Consider that we number these levels, starting from l_0 (the level that contains only one node, the node most far away from the outer border), until reaching the outer border at level l. If we assume that every inner node belonging to level l_i, $0 \leq i < \log N/\log\log N$, sends the value it stores and the values it has received from the nodes belonging to levels $l_j, j < i$ to its neighbors in the l_{i+1} level, clearly all the values stored by inner nodes 'go down' along the leveled graph, until they eventually reach a node that stops their path. Moreover, the number of times every value will be replicated at the outer border depends on how 'far away' is the node storing it. So, key 0000 is replicated $O(x^x)$ times, since its value has crossed x levels, and at every level the number of replicas is increased by a well defined factor; the node 1000 has x replicas instead, since it is directly connected with the outer border without intermediate levels.

If C represents the number of values that are cached in an outer border's node with this mechanism, the maximum value C_{max} experienced by an outer border's node i can be obtained considering the number of i's direct neighbors belonging to the hot zone plus the number of inner nodes reachable from i in at most $\log N/\log\log N - 1$ steps upward along the leveled graph: that is, we must recursively take into account its direct neighbors, the direct neighbors of these neighbors and so on.

Consequently, we can calculate the value of C_{max} as:

$$C_{max} = (\log N/\log\log N)! \in O(\log N/\log\log N)^{\log N/\log\log N}$$

- **$\mu = N/2$:** an outer border's node holds $\log N/2$ neighbors belonging to M, which, in turn, keep $\log N/2 - 1$ links directed toward lower level nodes and so on. The load C_{max}, in terms of items to cache, experienced by an outer node results:
$$C_{max} \in O(\log N/2)^{\log N/2}$$

In order to obtain a lower C_{max}, we can remove the assumption that an outer border's node does not forward any value. Instead, we can consider that every node, when its cache stores $\omega(1)$ values, divides this amount of values among its neighbors not belonging to the hot zone (or, in general, not in blocked state). In the model considered above, this strategy means that the load experienced by an outer node can be consequently shared among its neighbors, with a subsequent "expansion" of the outer border. This process is possible because, in every (realistic) network we consider, $\mu \leq N/2$ and, representing the hot zone as a leveled graph, all the nodes belonging to the first $\log N/2$ levels have more (or equal) links directed toward the lower levels of the leveled graph than toward the higher ones. This way, the load over a node is recursively divided, at every level, by the number of "lower links" every node belonging to that level maintains.

For instance, if $\mu = \sqrt{N}$, we "expand" the outer border with j additional levels, where j is a value such that

$$\frac{C_{max}}{(\log N/\log\log N + 1)(\log N/\log\log N + 2)...(\log N/\log\log N + j)} \in O(1)$$

that is, the cache size required for the node that stops the forwarding mechanism of cached values (and, consequently, for all the nodes above it) is constant. It can be easily seen that a value

$$j = \log N/\log\log N$$

can (amply) satisfy our request. The number of nodes B_{ext} that constitute the expanded outer border can be estimated as:

$$B_{ext} = \sum_{i=\lg N/\lg\lg N}^{2\log N/\lg\lg N} \frac{\lg N(\lg N-1)...(\lg N - i)}{i!} \in O(\lg N)^{\lg N/\lg\lg N}$$

If $\mu = N/2$, an extreme situation occurs: every hot node caches its keys in (approximately) one node not belonging to M.

The selection of the values to "discard" from a node's cache and to distribute among its neighbors could be purely random or more pondered; in general, it is opportune that a node maintains in its cache the values corresponding to its nearest nodes, because a path directed to one of them is more likely to pass through it. But, if we still adopt a randomized routing algorithm, finding *where* a requested value is cached could be more difficult.

Alternatively, a different approach may be chosen. Consider the worst case, in which $\mu = N/2$; if every node belonging to the hot zone is replicated only on a constant number of outer border's nodes, the "cache" load experienced is logarithmic. This fact can be easily proved.

Using the Stirling's approximation, we easily obtain that

$$\binom{2n}{n} = \frac{2^{2n}}{\sqrt{\pi n}} (1 + O(1/n))$$

where n represents an integer. In our case, the number of nodes belonging to the outer border is equal to $\binom{\log N}{\log N/2}$ (since we are considering the "median" level in the graph); so, the number of such nodes is estimated as

$$\binom{\log N}{\log N/2} = \sqrt{\frac{2}{\pi}} \frac{N}{\sqrt{\log N}} + O\left(\frac{N}{\log^{\frac{3}{2}} N}\right) > \sqrt{\frac{2}{\pi}} \frac{N}{\sqrt{\log N}}$$

Therefore, the "cache" load L_c results:

$$L_c \in O\left(\log^{\frac{1}{2}} N\right)$$

Unfortunately, even in this case, it is expected that the randomized routing algorithm is not able to find rapidly the node that stores the requested value within its cache.

A tradeoff, hard to balance, is found: if a great number of replicas is available, it's more likely that a lookup path can reach at least one among them; but the necessary cache size could grow faster than a constant. In contrast, if we keep a $O(1)$ cache size (as we have requested), the number of trials necessary in order to get the requested value increases excessively. The scheme just described fails in bringing together all the properties we want our overlay to have (see the Introduction): with $O(1)$ load and $O(1)$ cache size, the routing complexity can't be $O(\log N)$; vice versa, with $O(\log N)$ routing complexity, $\omega(1)$ cache size is obtained. In the following sections, an approach able to efficiently satisfy these requisites is developed.

4.3 Our overlay

Following the "two-tiers" model according to which a P2P network can be represented (that is, a structuring layer and a routing layer), we present both aspects treating them separately, in two different sections.

This section aims to provide a precise description of the structuring layer, that is, of the overlay we propose: subsection 4.3.1 specifies how objects and nodes are mapped into the identifier space and how each peer chooses its neighbors and organizes its routing table keeping it updated; subsection 4.3.2 explains how departures, failures or new arrivals of nodes are handled.

4.3.1 DHT organization

As in every DHT, every object available across the network is identified by means of a $< Key, Value >$ ($< K, V >$) pair. Each key and each participating node's IP address are then hashed into a common identifier space, where an ID is a binary string d bit long. The "*closeness*" between a given keyID and a nodeID in the identifier space is the parameter used in order to decide which nodes are responsible for that keyID, during a storing or lookup process: this concept will be more precisely explained in section 4.4.

Every node directly "knows" a limited number of other peers, storing their references within a *routing table* of size d: such stored peers represent the *neighbors* of that node. The way in which nodes choose their d neighbors should reflect the hypercubic-like organization of the P2P resulting network; in particular, the neighborhood relationship is formally defined as follows:

Definition 10. *Two nodes A, B are* neighbors *if and only if their Hamming distance equals to* 1 *($H(A, B) = 1$).*

Fully populated network For simplicity, we start considering how the network is organized when it is completely populated: indeed, if there is the maximum possible number of on-line peers N ($N = 2^d$), the network organization reflects exactly a d-hypercube where each node is a vertex, and each edge represents a "neighborhood" relationship. Figure 4.7 exemplifies the organization of a network with 3-bit long IDs, containing $2^3 = 8$ nodes.

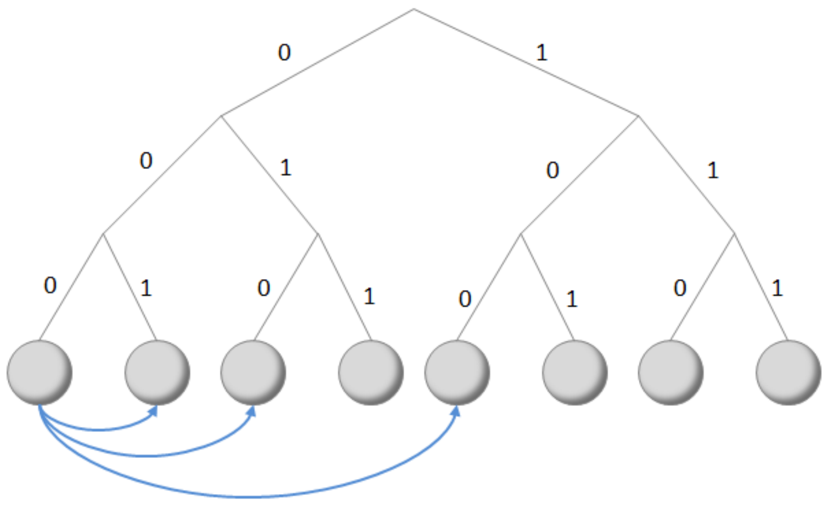

Fig. 4.7: A tree-like representation of our overlay: blue arrows indicate the neighbors of node 000.

Note how, in this case, we've adopted a particular representation for our hypercubic overlay, which opens a different "view" on the network; as we'll see, this novel perspective will be very useful when discussing the possibility of dynamic (leaving and joining) nodes and routing and caching mechanisms. In such a representation, the nodes are organized using a *binary tree* (which is not necessary complete as in this case, but possibly well balanced): each leaf of the tree corresponds to a node, which is assigned, as nodeID, the binary string describing the path from the root to the leaf itself. According to our convention, we assume that left branches are represented by a 0 and right branches are represented by a 1. Therefore, tree edges are used exclusively in order to define the ID corresponding to each leaf; blue arrows don't represent physical links, but "logical" links instead, meaning a "neighborhood relationship". Here, in particular, the three neighbors of node 000 are shown.

Since A's routing table stores the addresses of all the neighbors of A, it's easy to define its entries: indeed, the d entries stored by A are the d IDs obtained by flipping one bit of A's ID at a time, starting from the most significant: for example, if $A = a_0...a_i...a_{d-1}$, entry 0 corresponds to the node with ID $\bar{a}_0...a_i...a_{d-1}$, entry 1 to the node with ID $a_0\bar{a}_1...a_{d-1}$ and so on.

Note that, if there exists an arrow between node A and node B, this means that A and B are neighbors, so A stores B's address in its routing table, and vice versa (a double arrow symbol would be correct as well).

In fact, in this case, differently from Kademlia, the neighborhood relationship is *bidirectional* and *symmetric*: that is, if a node labelled as A stores within its routing table a contact B as its *ith* entry (therefore, $A = a_0...a_i...a_{d-1}$ and $B = a_0...\bar{a}_i...a_{d-1}$), B surely stores A's references as its routing table's *ith* entry. Thanks to this observation, we can introduce the concept of *sibling* nodes.

Definition 11. *Two nodes are* siblings *if their IDs differ only in the least significant bit and, therefore, the one is the last entry within the other's routing table.*

Consequently, in this configuration, each node has a unique sibling, and $d-1$ additional neighbors. The height reached by the tree is equal to d; this is, in particular, the maximum possible value.

Note also that two nodes that are siblings according to our definition are siblings also according to the usual terminology adopted for tree graphs: for example, nodes 000 and 001 are siblings, and the height of the smallest subtree containing them is 1. If we consider a node, say 000, and identify all the smallest subtrees containing it, starting from subtree with height 1 and proceeding until reaching the subtree with height d, we'll see that each one of such subtrees contains exactly one neighbor of 000, placed in the opposite half of the subtree with respect to 000 itself.

Partially populated network The description we've just provided is quite simple, but it is not sufficiently detailed. In fact, it may be possible that we are not dealing with a complete binary tree (and, in real environments, it is often the case).

Consequently, the hypercubic structure establishing the neighborhood relationships needs to be *generalized.*

First of all, we provide the definition of a node's level:

Definition 12. *The* level *of a node corresponds to the number of digits constituting the node's ID in the generalized representation; in the tree-like representation, it corresponds to the depth owned by the corresponding leaf.*

While in a fully populated network all nodes are located at the same level, in a partially populated network leaves can be found at different levels. In particular, the difference between two nodes' levels (that is, the difference between their ID's lengths) represents the *gap* existing between those nodes. Figure 4.8 shows the configuration that would be assumed by the network represented in Figure 4.7, if nodes 010 and 011 had not joined the network: in Figure 4.8(a) the tree-like representation is provided, while Figure 4.8(b) illustrates how the corresponding hypercube graph can be thought of.

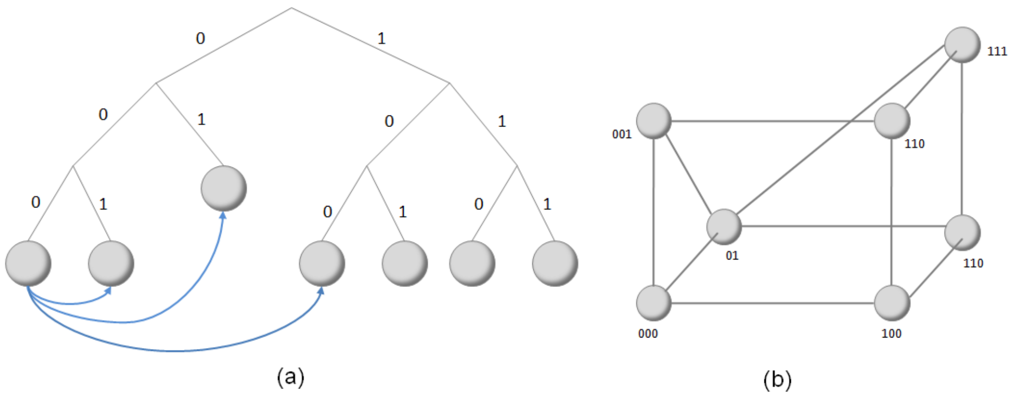

Fig. 4.8: A tree-like representation of a partially populated network.

At this point, it must be accurately defined how neighborhood relationships are decided in this case.

According to the approach described in [2], a node corresponding to leaf A maintains a pointer to the node at leaf B if B's ID is a prefix of a string obtained from A's ID flipping one bit in A's ID and appending it with zeros. However, note that, in this way, links become *not bidirectional*: nodes with shorter ID have a routing table with less entries. For directly preserving the hypercube structure, we maintain the bidirectionality of links, adopting a scheme similar to that described in [1].

A node n, which in a fully populated network would have node m as neighbor, establishes the corresponding contact according to these rules:

- if m actually belongs to the network, it is chosen;
- if m does not belong to the network, but an ancestor p of m does (that is, p is a leaf whose ID correspond to a prefix of m's ID), n chooses p as its neighbor;
- if m does not belong to the network, but there exists a set S of m's descendants (that is, m's ID corresponds to a prefix of these descendants' ID), n chooses *all* the nodes belonging to S as neighbors.

Figure 4.9 exemplifies these rules.

Clearly, particular attention should be paid to the maximum gap existing between a node and its neighbors; otherwise, in highly unbalanced trees, routing table's size could become excessive.

Finally, we provide a conclusive note. The mechanism evolves in such a way that IDs of different lengths are assigned to nodes; note, however, that this is

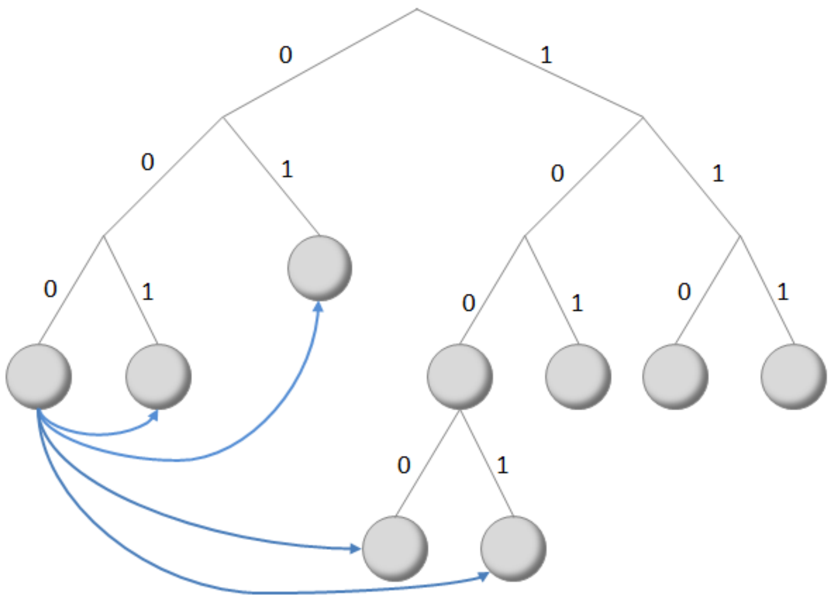

Fig. 4.9: Neighborhood relationships in partially popualted networks: blue arrows show the contacts owned by node 000.

a simple convention introduced for the sake of clarity: a node with "truncated" ID simply covers the zone that all its children together would have covered. For example, node 00, in a network of maximum size 8 (i.e., a 3-hypercube), can "potentially" become node 000 and 001, so, as long as no node arrives and covers (one between) that positions, 00 must act in place of both of them. One could also, more formally, think of node 00 as having a "standard" ID, say 000, and being responsible for its location and for location 001 as well: simply, the "truncated" IDs utilization offers us a simpler and more immediate representation. In the following, this convention will be always adopted: we'll refer to such a representation as the "generalized representation".

4.3.2 *Joining and leaving nodes*

The scheme just described works only in a static environment. In order to appropriately handle *dynamic* situations, it's necessary to define the network's behavior in case of *transient* peers (that is, peers that can "appear" or "disappear" at every moment): it is the problem this subsection focuses on.

When dealing with a dynamic environment, the main goal is to maintain, somehow, a constant, underlying hypercubic structure. Intuitively, this works by *splitting* a node (or, more correctly, its location) when a new node joins the network, and, inversely, by *merging* two nodes (their locations) where a peer leaves the network. This process is well visible if applied to the tree-like representation: a single-node network corresponds to a tree containing only its root node; if there are k participating nodes ($1 < k < N$), the corresponding tree contains k leaves, eventually placed at different levels: tree's inner vertexes represent nodes that no longer exist (they were split), the leaves represent current nodes.

In order to guarantee an acceptable load balancing among the nodes, in terms of the number of keys a node is responsible for, as well as a good lookup performance, the locations to split or merge should not be chosen in a totally random fashion. In fact, this would lead to a highly unbalanced structure: referring to the tree structure described above, some leaves would be placed at a much deeper level than others. Our goal is to keep the tree balanced at all times, that is, to minimize the gap among nodes. Ideally, we would like the structure to be able to dynamically "readapt" itself, in such a way that, if it contains l nodes, with $0 < l \leq 2^d$ and $2^h \leq l \leq 2^j$ for two integers h, j $(0 \leq h, j \leq d)$, all the IDs owned by active nodes correspond to a binary string of length at least h and at most j. In other words, we would like to keep the gap between any pair of nodes as small as possible, in order to still achieve a diameter of $\Theta(\log N)$.

The arrival and departure handling mechanisms pursue exactly this goal: we provide a detailed description for both of them.

Joining nodes An arrival is accomplished by first finding an appropriate leaf of the tree (the mean of "appropriate" will be clearer in the following), and then performing a *split* operation on the leaf itself. A split of leaf i replaces it with an internal node plus two children of that node; the node that previously resided at leaf i is assigned to one child, and the new arrival is assigned to the other child. The crucial point is the choice of the leaf to split, in such a way that an acceptable load balancing is maintained.

Whenever a node C needs joining the network, it must know the IP address of *at least* one node, say A, already belonging to the overlay (this is a *bootstrap* requirement common to almost all P2P environments). After receiving such a join request, A follows a well defined *arrival procedure.*

Once it has been contacted, A start considering its neighbors and, as long as it can follow a path towards lower level nodes, it follows that path until a node B is reached, with no lower level neighbors. Then, node C is splitted into two children: B itself and the new node C. Figure 4.10 shows an example.

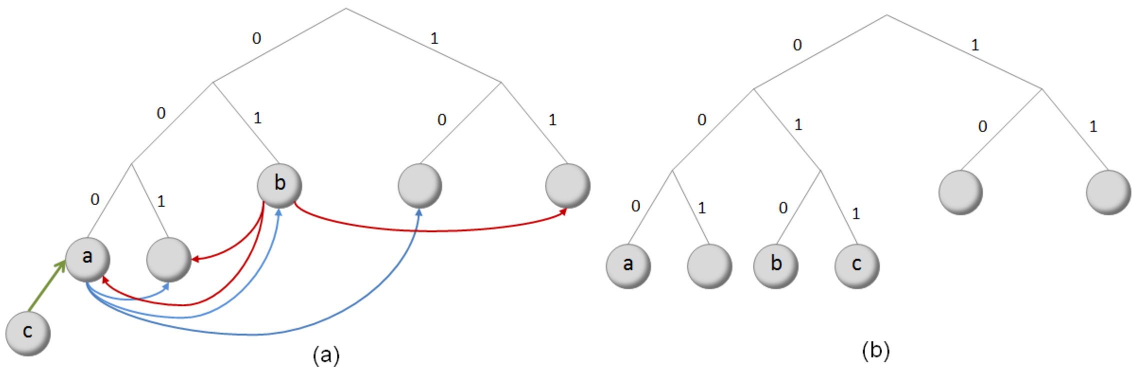

Fig. 4.10: An example of arrival procedure.

In Figure 4.10(a), node c contacts node a in order to join the network; a stores in its routing table b, that belongs to a lower level. Since b does not hold, in turn, lower level contacts, the search process stops, selecting b as the node to split. So, b's level is increased by one, and c and b become siblings (Figure 4.10(b)).

As the last step, the new node may efficiently locate its neighbors, since it already knows its sibling: through the sibling's routing table, it easily fills the remaining entries in its table with the appropriate neighbor's data.

Leaving nodes Often, a node voluntarily leaves the network: in this case, it disconnects itself by the overlay following a preestablished procedure. In other cases, it may happen that a node suddenly fails, without sending their neighbors any signal. In order to become readily aware of the existence of such a situation, nodes periodically exchange PING messages with their neighbors. If a node does not answer within a predefined time interval, the neighbor waiting for the reply message notices that a failure has probably occurred, so it starts a *departure procedure*.

Define the node that is leaving the network as A. The procedure can start at A itself, or at one of its neighbors, B, once B has pinged A without receiving any reply. Begin at one of these nodes, as long as there is a direct link toward a higher level node that link is followed. When the process stops, two sibling S_1 and S_2, with no higher level's neighbors, are found. Consider that S_1 and S_2 are children of vertex v. S_1 replaces node A, while S_2 change its location to v.

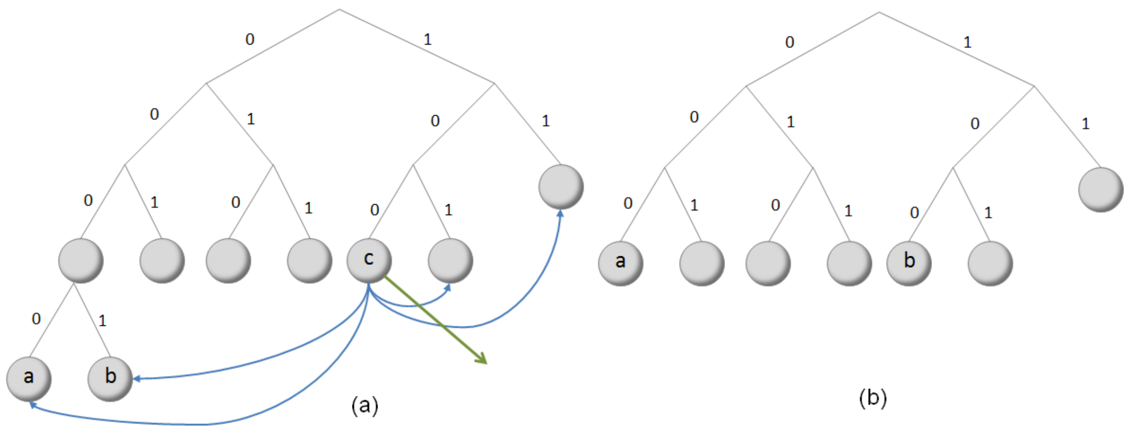

Fig. 4.11: An example of departure procedure.

In Figure 4.11, this kind of situation is illustrated. Node c wants to leave the network; it stores in its routing table the contacts of two sibling nodes, a and b, located at an higher level (namely, the highest level owned by any node in the tree). So, b is chosen for replacing c: it is assigned c's old ID, 100; a's ID is "truncated", from 0000 to 000. The old locations corresponding to a and b are merged.

Note that the two operations (departure and arrival) are, substantially, *symmetric*.

Furthermore, any arbitrary sequence of arrivals and departures keeps the network balanced. In particular, the following theorem holds:

Theorem 6. *Given any sequence of arrivals and departures, the gap among any pair of neighbors is guaranteed to remain* 1 *at all times.*

Proof. The key idea is that nodes that get split (or merged) are in a *locally* minimal (or maximal) level. The proof can be conducted by induction:

- Basis: consider a network of size 2: we have two nodes, both children of the root node. If a new node joins the network, the difference between the lowest and the highest level in the tree is not greater than 1.

- Inductive step: suppose that there are n active nodes. Due to the algorithm described above, the node chosen to be splitted (or merged) is at a minimum (or at a maximum) with respect to *all* its neighbors. So, the gap between the modified node and its neighbors remains unchanged.

□

Corollary 2. *The maximum gap among any pair of nodes is* $O(\log N)$*.*

Finally, it's necessary to estimate the overload produced by these "rebalancing" adjustments. In the worst case, the number of nodes examined during rebalancing is equal to the maximum difference between any two nodes in the network ($O(\log N)$).

As we've already underlined, this mechanism, addressing the problem of maintaining as balanced as possible the network's structure, has been introduced mainly for two reasons: first, such a structure ensures that nodes are equally distributed over the identifier space, in such a way that no peer is responsible for much more keys than others peers; second, great differences among nodes' levels should be avoided since they dramatically affect routing's efficiency.

This last problem is the object of the following section.

4.4 Our routing algorithm

Keeping the structure just described in mind (mainly, its tree-like representation), it's possible to establish a relationship among nodes and keys based on their location in the identifier space.

The assignment is, in some regards, similar to that adopted by Chord, but with the difference that, while in Chord the identifier space is circular, here that space spans along a straight line. If we denote with ID_i the binary identifier assigned to the *ith* leaf ($0 \leq i \leq l-1$, where l is the number of participating nodes), considering to start from the left, the node corresponding to ID_i is responsible for storing all the objects that are mapped into the interval $[ID_i, ID_{i+1})$. In addition, the node corresponding to the first leaf is also responsible for the interval $[0, ID_0)$ and the last leaf for the interval $(ID_{l-1}, N-1]$. In the tree-like representation, one can consider that every leaf is responsible for all its descendants leaves, that would exist in a fully populated network (Figure 4.12).

Clearly, such a correspondence defines a precise, unambiguous rule according to which each peer performs storing or lookup operations: whenever a node wants

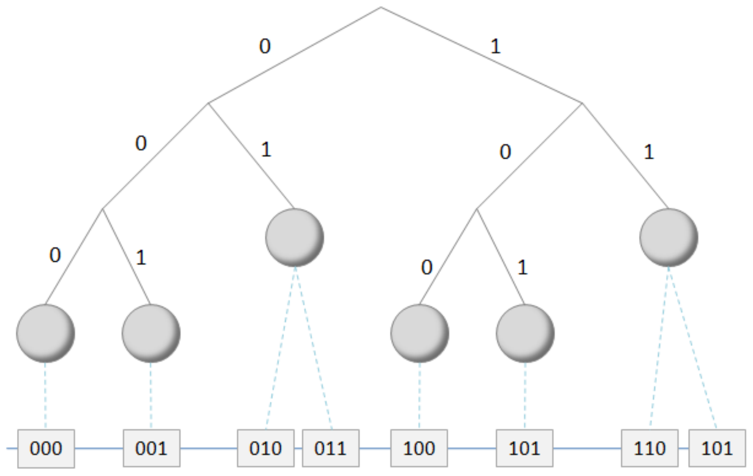

Fig. 4.12: Partitioning of the identifier space among current nodes.

to store or search an object, it hashes the object's key and, then, performs a node lookup.

Therefore, the lookup primitive, accomplished by means of a well defined *routing algorithm*, can be considered the "foundation" other operations are built on.

Note that there exists a fundamental property owned by hypercubic network structures, that coincides with one of the requisites we were looking for in order to develop our routing and caching mechanism. This basic characteristic is the *routing flexibility*: the hypercube can use *out-of-order bit fixing* since a node's neighbor only differs from itself on a single bit and hence *previous corrections* of lower order bits are *maintained* as higher order bits are corrected. In particular, considering two distinct nodes A and B such that $H(A, B) = k$, there are a total of $k!$ different paths among them, *all* of minimal length l. In fact, the order by which different bits are exchanged hop by hop can be arbitrarily chosen. Furthermore, recall that, in a d-hypercube, d parallel of lenght at most H paths exist among any pair of nodes

Incidentally, it's exactly due to this characteristic that hypercube graphs have been widely studied in parallel machine environments: this wide range of choices offers robustness and fault tolerance.

This is the main difference that distinguishes the hypercube from other overlays, for example *Kademlia*. Kademlia, as we've already shown, employs a tree-like overlay organized by means of a XOR metric; according to this structure, ID's bits must be corrected in *strictly left-to-right order*: otherwise, the correction of a lower bit could cause an higher bit to get wrong, compromosing the algorithm's final convergence. Actually, under failure, some Kademlia implementations adopt a less rigid scheme: even if a node cannot fix the highest different bit, it can still make progress in the XOR distance, effectively fixing a lower order bit. Thus, different paths can be chosen, but these paths are not of equal lengths. Intuitively, even though Kademlia offers the flexibility of fixing lower order bits before higher ones, the lower order fixed bit need to be preserved by later routing hops that fix higher order bits.

As noted in [14], a natural drawback of the hypercube's routing adaptability is the lack of flexibility in *neighbor selection.* A node does not have a set candidate neighbors, among which it can select its direct contacts: each one of its neighbors is strictly predetermined. Whereas, as we've already seen in the previous section, this does not translate into routing table's updating problems, complications could arise when one introduces *latency* considerations. That is, high latency could be introduced if predetermined neighbors are geographically far away. This fact must be taken into account and, in widely deployed environments, adequate policies (for instance, targeted node placement strategies) should be adopted in order to mitigate subsequent negative effects (as delays).

As we've anticipated, the core of this work is the attempt of *unifying* searching and caching, two distinct sides of the same routing mechanism. Various challenging aspects must be taken account. In particular, the main problems our technique faces are the following:

- *Search efficiency*: the routing algorithm we adopt should preserve a $O(logN)$ (or, at most, *poly* $\log N$) hop long path, which is guaranteed by most DHTs; clearly, with the utilization of a caching scheme, this asymptotic bound is expected to eventually improve, not certainly to become worst. But remember that one of our main goals is to avoid *every node* to be *overloaded*; guaranteeing it inevitably means causing an overhead for the whole network. So, we would like such a overhead not to affect dramatically lookup efficiency: the average lookup hops achieved by a P2P overlay is one of the most important factors to measure the performance of P2P systems, useful to diminish the request latency in applications using them;

- *Replicas full utilization*: in order to avoid resource consumption, we would like replicas to be fully utilized: that is, replicas should be placed at nodes that are likely to be visited during a lookup process;

- *Replication refresh*: it's a tricky problem. In dynamic environments like P2P networks, contents are continuously provided and removed; it would be necessary to constantly check cached values consistency, in order to keep them up-to-date. For example, due to this difficulty, some Kademlia-like DHTs (like Kad) do not implement any caching mechanism at all and try to keep load balancing only throug proactive replication schemes;

- *Resources consumption*: there exists a fundamental tradeoff between replication and resource consumption. More replicas will generally improve lookup performance, but at the cost of storing space and bandwidth consumption. In fact, the exchange of cached values causes additional traffic, and a part of the storing capacity available at every node must be reserved for caching purposes. So, if we would like the number of each item's replicas to be just as little as necessary in order to guarantee acceptable lookup performance.

Our lookup and caching strategies address that problems, achieving a satisfactory compromise.

4.4.1 Lookup operations

In our approach, the routing algorithm can be thought of as *recursive*.

Recall that, in an *iterative* lookup, a node that is requested to look up a key will return the network address of the next node found to the requesting node; this node will then request that next node to take another hop in resolving the desired key. The Kademlia routing algorithm is clearly an example of iterative lookup. In contrast, in *recursive lookups* a node simply forwards the query just received to the next node. The query should carry information about its sender, in such a way that the target node, once reached, can directly provide the requesting node with the desired value.

Whenever possible, during a lookup process ID's bits are adjusted "from left-to-right"; sometimes, this conventional mechanism can't be accomplished, since some links could fail or be "blocked" (as we'll explain later in this chapter). In these cases, the choice of which bit, among the different ones, should be corrected at next hop is made in a totally random fashion.

Figure 4.13(a) show how lookup search evolves in a standard and intact network, while Figure 4.13(b) illustrates how this search *could* evolve if some links are not practicable.

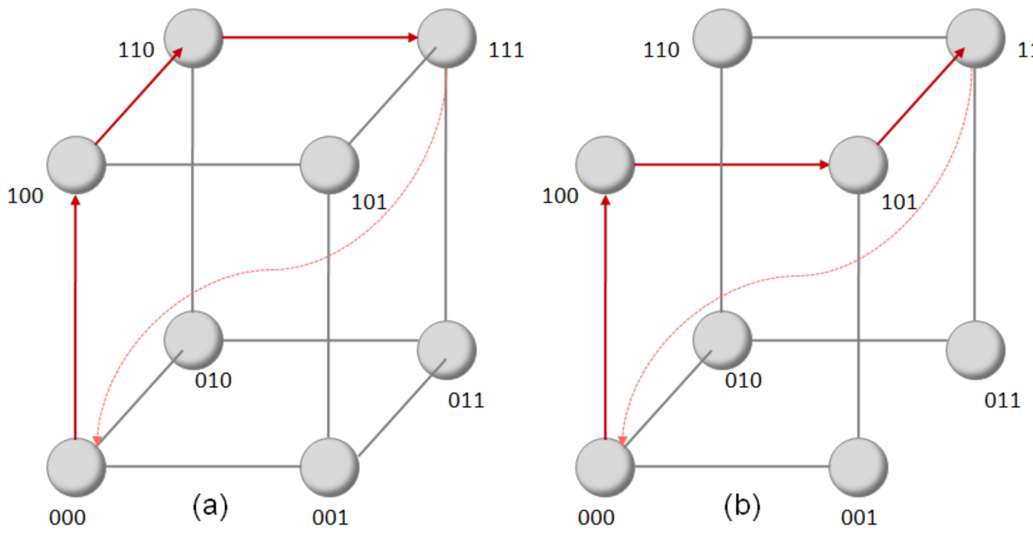

Fig. 4.13: Lookup path from node 000 to node 111 when all links are available (a) and when some links are not practicable (b).

Note that it is possible, in some cases, that one or more nodes become unreachable. In order to prevent the occurrence of this fact, well defined countermeasures are adopted. Since these countermeasures are strictly related with our caching mechanism, we address this problem in the following subsection.

4.4.2 Caching mechanism

First of all, let us provide a preliminar clarification: no proactive replication is prescribed by our protocol. Therefore, only *one* copy per-object is initially stored;

additional replicas are created, if necessary, in a successive moment. However, the introduction of such an aspect is not problematic: its effective utility can be evaluated according to the contingent situation (i.e., a well defined, implemented network).

We've studied two different types of caching mechanism: both are evaluated against an *adaptive online adversary.*

Adversary model This adversary model is taken into account: our caching algorithm and the adversary take turn. At the adversary's turn, it can choose a (never used before) node as the *requesting node*, and an arbitrary target node, issuing a query for it. At the algorithm's turn, it may use some computation and message passing. In particular, the adversary aims to reach *at least* one of the following goals:

- the load experienced by at least one node in the network is $\omega(\log N)$;
- the worst-case lookup path length is $\omega(\log N)$;
- one or more objects become unreachable.

In contrast, our caching algorithm must guarantee that *no one* among those situations occur: we call it our *main requirement.* Ideally, we would like the adversary to be able to choose up to $N^{1-\epsilon}$ requesting nodes. Let us analyze in details how our algorithms work.

First caching algorithm The first caching algorithm we propose is a very simple one: when the load experienced by a node exceeds a predefined threshold (which, in our case, corresponds to $O(\log N)$), this node acts exactly as if it were leaving the network. In other words, that node starts the departure procedure we've already analyzed above, storing the value it is responsible for within the chosen substitute's cache.

The analysis we've already carried out for the departure procedure can be easily referred to in order to show that our main requirement is fulfilled. Note that, however, this represents a simple mechanism avoiding hotspots, not an authentic caching mechanism: in effect, even if it works well against the adopted adversary model, it never creates new replicas of an object. So, in usual conditions (that is, when the network is not attacked), it is expected to offer a worst performance with respect to the other, standardized, caching strategies we've mentioned in Chapter 2.

Second caching algorithm Our second approach leans on the *recursive structure* owned by hypercube graphs. Indeed, every node is considered as belonging to $\log N$ hypercubes of increasing size (we define these graphs as "*local* hypercubes"): the smallest is constituted by the node itself and its sibling (it represents a 1-hypercube); the greatest hypercube is the network itself.

Each node n constantly monitors the incoming traffic, considering the amount carried by each distinct link. When the traffic over one of these links exceeds a predetermined threshold (recall that we want to guarantee a logarithmic per-node load), that link is blocked. Consider that it is the *ith* link (i.e., the one between n and the neighbor whose ID differs in the *ith* bit). This means that *every* query aiming to *change* the *ith* bit is blocked: n is actually *denying* messages' access into a well defined *section* of the hypercube. Note how this represents a very flexible solution in order to mitigate the *routing* load: if a node is located at an overloaded traffic junction, it can decide to "deviate" the traffic it is not able to efficiently manage.

Since we permit a *randomized* routing algorithm, whenever a query can't pass through a certain link, it has *flexibility* enough in order to choose an alternative route.

In order to keep an acceptable data availability across the network, some constraints are defined:

- Rule 1: links connecting two siblings n and m can't be broken. As soon as the traffic coming from its sibling becomes excessive, both n and m must cache the values they are responsible for in the two nodes s and t that, together with m and n, constitute the *smaller* local (2)-hypercube of *greater* size (s and t differ respectively from m and n in the penultimate significant bit);

- Rule 2: nodes belonging to the same local hypercube periodically check how many links connecting the local hypercube itself to the "outside" are still available. If more than half of such links are blocked, all the nodes belonging to the local hypercube cache their data in the the *smaller* local hypercube of *greater* size (in particular, each node is replicated in its own neighbor);

- Rule 3: if the *smaller* local hypercube of *greater* size just mentioned can't substain that additional load, its nodes, in turn, immediately cache their values in the "next" local hypercube.

Note that, according to this mechanism, a node can repeatedly cache the values it is responsible for in all its neighbors. The strategy guarantees that, in every case, a logarithmic per-node load achieved with constant cache size; the routing efficiency will be evaluated in the next section.

A final remark should be made: constraint 2 requires that every node is aware of the state of *each* local hypercube containing the node itself. In order to achieve this goal in a distributed environment, a simple mechanism could be implemented: whenever a node blocks a link, it broadcasts a signal to all the interested nodes ("interested" means belonging to the local hypercube whose connectivity with the outside is affected by that operation: it can be individuated considering which is the blocked link). Other, more efficient, distributed algorithms could be applied as well.

4.5 Performance analysis

In this section, we provide a quantitative analysis of the overlay just proposed, taking into account three distinct aspects: load balancing (mainly in terms of partitioning of keyIDs among active nodes), routing (that is, lookup performance: this value directly determines also storing operations complexity) and, finally, caching (especially trying to estimate how "blocked" links can affect routing efficiency).

Load balancing Note that, thanks to Theorem 6, the number of keyIDs every node is responsible for is at most *twice* the number of keys assigned to anyone of its neighbors.

Routing First, let us consider the lookup efficiency achieved without taking into account any cachimg mechanism: in this case, an easy and immediate reasoning can be argued. Indeed, since, once again, Theorem 6 and Corollary 2 hold, considering the "left-to-right" adjustment of ID's bits accomplished during the lookup process, we obtain that an upper bound for the maximum number of routing hops is still $O(\log N)$.

If we introduce the first caching mechanism, the argumentation does not differ. As we've already seen, the overhead produced across the network is asymptotically logarithmic in the worst case.

Analyzing the second caching mechanism is a more challenging problem.

First of all, note that, thanks to the three rules described above, the graph recursively keeps its *connectivity*. In order to see it, one may pick an arbitrary node n: due to rule 1, it is surely connected to its sibling m. Due to rule 2, at least one of those nodes is connected with a node whose ID differs in the penultimate bit and so on.

The *expected* number of routing hops can be easily calculated: since every node is obligatorily connected with its sibling, once we reach a node, we surely are able to reach also its sibling. Therefore, since each local hypercube maintain available at least half of the links connecting it with the other nodes, it is expected that, if we choose an arbitrary bit to adjust and an arbitrary node n, at least one between n and its sibling lets us change that bit, without blocking the lookup path. Consequently, the expected number of routing hops between any pair of nodes is estimated as $2 \log N$.

But, what is the guaranteed performance if the adversary chooses a requesting node and a target node connected through a very "winding" path? An intuitive argument is the following: consider a node n, with ID $n_{ID} = n_0 ... n_{\log N - 1}$, aiming to reach node m, with ID $m_{ID} = \bar{n}_{ID}$ (that is, m is the most distant peer). n starts adjusting its ID's bits from left to right: doing it, it progressively "moves" into hypercubes of decreasing size, starting from a $\log N$-hypercube until reaching a 1-hypercube.

Then, we try estimate how many steps, in the worst case, are necessary in order to flip the ID's most significant bit in a generic d-hypercube. Consider two concrete examples: a 2-hypercube and a 3-hypercube. Figure 4.14 shows the worst admissible configuration.

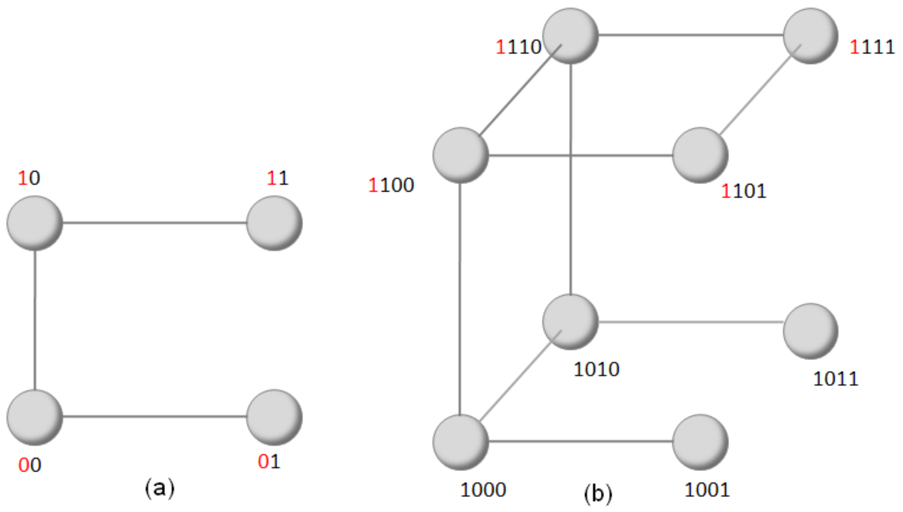

Fig. 4.14: Worst case configuration for a 2-hypercube (a) and a 3-hypercube (b).

In the worst case, a node needs (respectively) at most 2 and 3 hops in order to flip the most significant bit of its ID: these hops can be thought of as the *cost* of passing into the opposite half of the hypercube, that is, the half characterized by a different most significant bit (recall the hypercube graphs' recursive construction). Generalizing this observation for any d-hypercube, so that an upper bound of d hops is derivated, is not straightforward. One can consider that the greater is the number of "visited" nodes, the higher is the probability of discovering a node that is actually connected with the half of the hypercube we want to reach. In this way, the total number r of routing hops covered during a worst-case lookup would be bounded as:

$$r \leq \sum_{i=0}^{\log N-1} i < \log^2 N$$

This means that the estimated routing complexity is $O(\log^2 N)$. Nevertheless, a formal proof should take into account other, subtle, details, as the possibility of entering "dead-end" paths that inevitably trigger a "backtracking" mechanism.

4.6 Conclusions

In this chapter, a novel DHT overlay has been proposed and widely analyzed. Our overlay presents many favorable characteristics, as flexibility, robustness and load balancing.

A detailed specification has been provided for the most common DHT operations, like objects lookup and storing, or nodes arrivals and departures: the logarithmic performance achieved by all these operations is similar to that of other DHTs.

In addition, two different caching mechanisms easily implementable by our overlay have been suggested. While the first effectively satisfies our requirements, the second seems particularly attractive and would deserve further development, especially in order to define a formal upper bound on the routing load. We think that this approach exhibits good flexibility, so that many improvements could be obtained. For instance, a possible strategy could consist in assigning different *priorities* to query and links, so that queries are more likely to pass through a certain link after many resendings.

5. SIMULATIONS

This chapter provides some experimental results, in order to validate the theoretical results we've illustrated in Chapter 3: in particular, our simulations focus on the estimated lower bound for the "k-buckets network model" and the "Kademlia-like" model. There exists a specific reason for which the "1-buckets network model" is not taken into account: after some tests were run, network's behavior in such a context turned out to be very similar to that held by the "k-buckets" model, as we could expect, since the latter represents a generalization of the former; so, the experimental results obtained in both cases can be reasonably considered to dovetail.

Section 4.1 describes the test environment we have utilized and illustrates some significant results; on the basis of this results, some considerations are carried out in section 4.2.

5.1 Experimental setup and main results

In order to execute the tests, we have developed our own simulator: it is a program, written in Java language, that exactly fulfills our requirements. Whereas many P2P simulators are available nowadays (for instance, PeerSim[1] and OverSim[2] were initially considered), they have been discarded as not very flexible, and hardly adaptable, tools: while they provide many advanced features, generally no one among them implements any caching mechanism.

Our program lets users set various parameters: N (network's size), k (bucket's size), c (cache size), q (recall that $q = \mu/c \log N$, where μ represents the hot zone's size), other additional values, which will be mentioned later in this section, and, of course, the specific model to simulate.

K-buckets network model For the k-buckets network model, a DHT similar to Kademlia has been implemented: each node's routing table is filled with contacts that are chosen in a random fashion, not violating the already described Kademlia rules. The implemented routing algorithm exactly reflects the one implemented in Kademlia. Every node is assigned a counter: whenever a node is contacted, it increments its own counter by one. The adversary is constituted by the $N/2$ nodes located in the opposite half with respect to the hot zone: adversary's nodes take turns at choosing a random target, belonging to the hot zone, and issuing a query for it. At the end of the entire process, we sum up the number of times each "hot" node has been contacted: this value, divided by the hot zone's size,

[1] peersim.sourceforge.net

[2] oversim.org

is reported as the *load* experienced by the hot zone itself. Note that, in this way, we take into account request load and routing load as well.

In the following, we present the results we've obtained on the basis of the following parameters:

- N: powers of two in the interval $[1024, 65536]$;
- $k = 10$;
- c: integers in the interval $[5, 20]$;
- q: integers in the interval $[2, 4]$.

Experiments' outcomes are summarized by means of simple charts: while the x-axis represents the network's size, the y-axis corresponds to the load experienced by the hot zone; an arithmetic mean, based on the different values of c we've considered, has been calculated. In every chart, the function $N/2q \log N$ is drawn too, so that a direct comparison can be made (recall that, after all, we are interested in the *asymptotic* bound) between the function itself and our results.

The following figures illustrate the results we've obtained for this model.

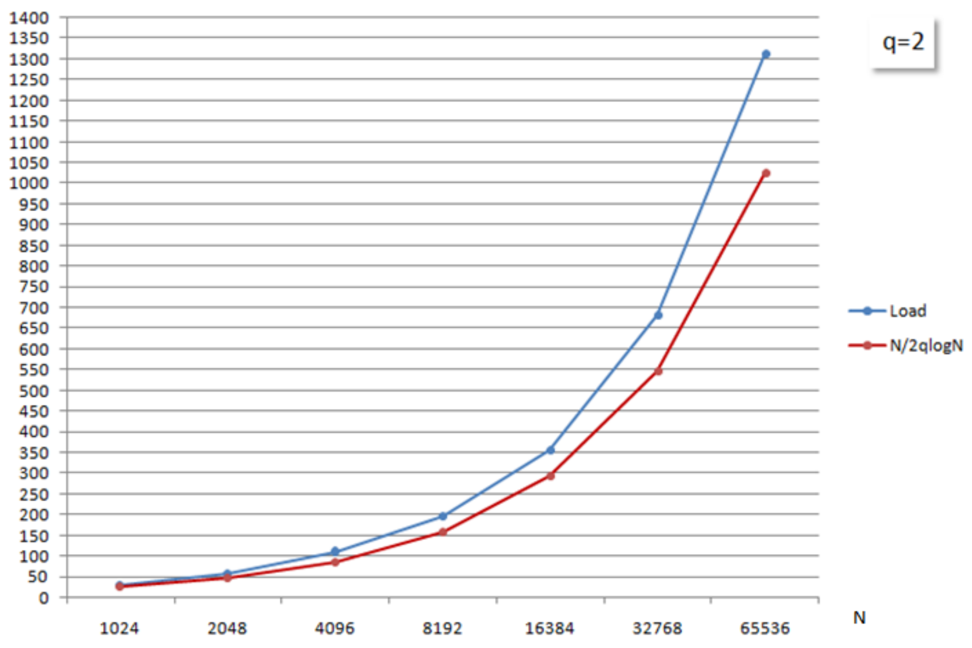

Fig. 5.1: Load experienced by the hot zone, compared with $N/2q \log N$ (K-buckets network model, $q = 2$).

Kademlia-like model As in the previous case, each node's routing table is filled with contacts that are chosen in a random fashion, not violating the usual rules; the implemented routing algorithm exactly reflects the one implemented in Kademlia; every node is assigned a counter: whenever a node is contacted, it increments

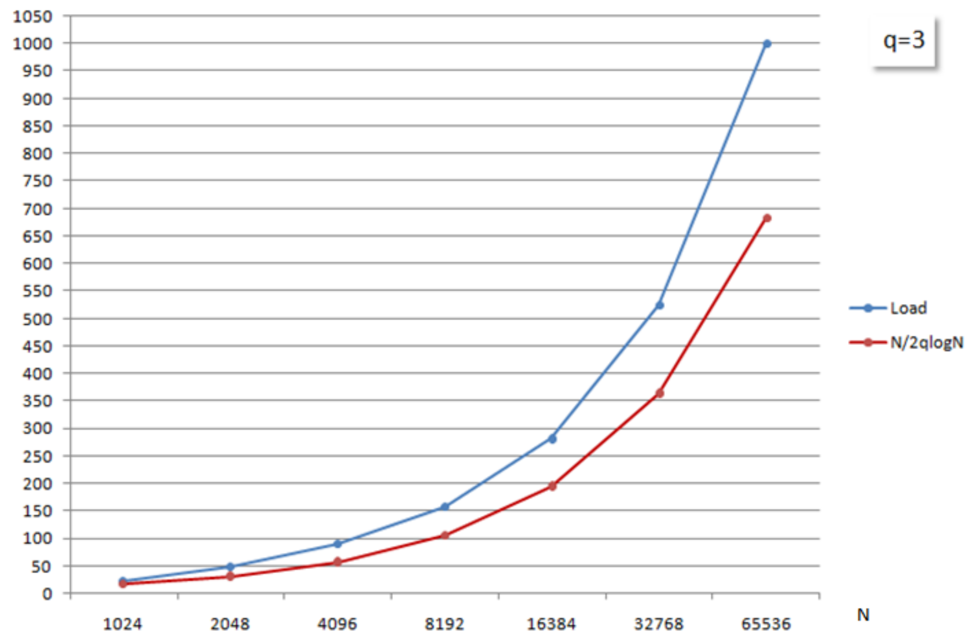

Fig. 5.2: Load experienced by the hot zone, compared with $N/2q \log N$ (K-buckets network model, $q = 3$).

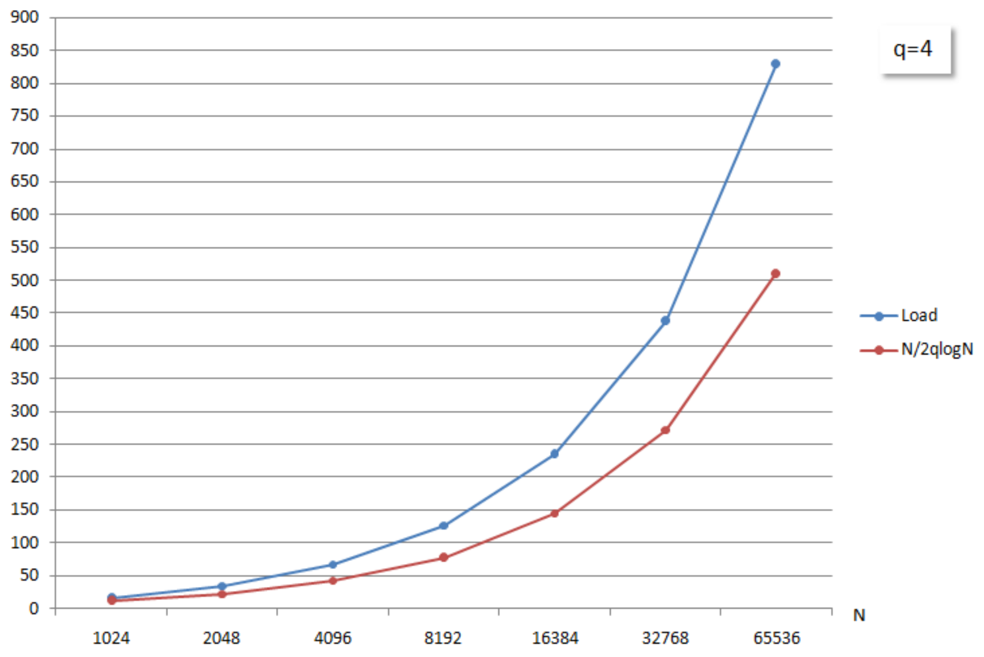

Fig. 5.3: Load experienced by the hot zone, compared with $N/2q \log N$ (K-buckets network model, $q = 4$).

its own counter by one. The adversary is constituted by the $N/2$ nodes located in the opposite half with respect to the hot zone: adversary's nodes take turns at choosing a random target, belonging to the hot zone, and issuing a query for it.

Differently from the previous case, here the load already experienced by neighbors is taken into account when choosing the next routing hop: in particular, nodes that exhibit *lower* counter's values are preferred and, if a node's counter exceeds a preestablished *threshold* t, that node, if not belonging to the hot zone, goes down (that is, it becomes unreachable). At the end of the entire process, we sum up the number of times each "hot" node has been contacted: this value, divided by the hot zone's size, is reported as the *load* experienced by the hot zone itself.

In the following, we present the results we've obtained on the basis of the following parameters:

- N: powers of two in the interval $[1024, 65536]$;
- $k = 10$;
- c: integers in the interval $[5, 20]$;
- $q = 3$; indeed, as we'll see, the experimental outcomes are very similar to those obtained in the previous case: so, would have been somehow "pleonastic";
- $\alpha = 3$ (according to the protocol's specification);
- t= 20.

Figure 5.4 illustrates the results we've obtained for this model.

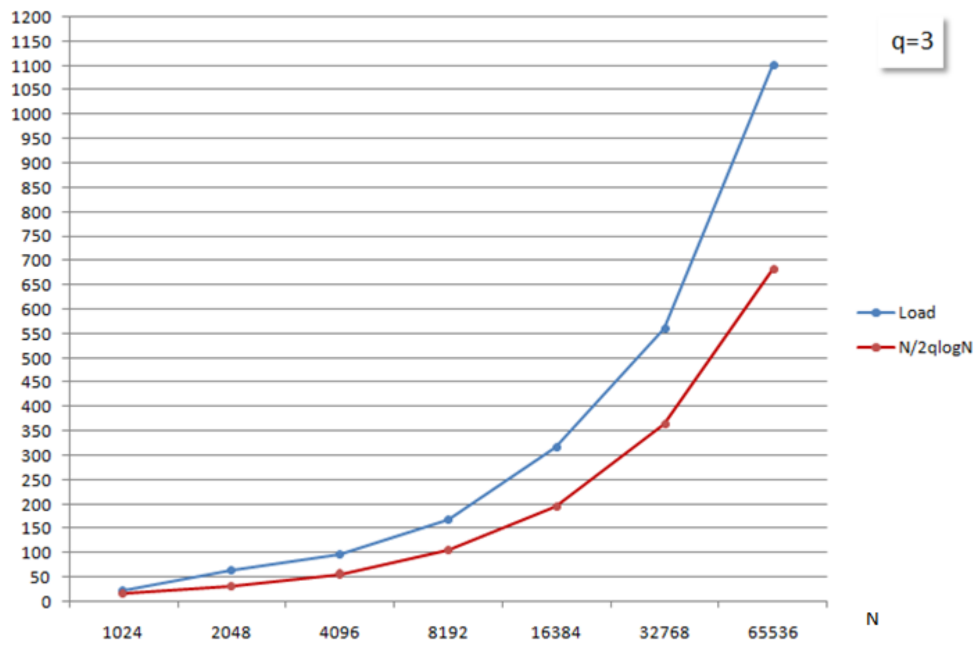

Fig. 5.4: Load experienced by the hot zone, compared with $N/2q\log N$ (Kademlia-like model, $q = 3$).

5.2 Results evaluation

As it can be easily seen, the experimental results we've obtained seem to validate our theoretical analysis.

In particular, note how, in all the examined cases, the *average load* experienced by hot nodes is higher than the standard function $N/2q \log N$. Furthermore, the gap between the two seems to increase in proportion to the network's size; debugging our program, in order to analyze when ayn node was mor likely to be contacted, we have seen that these effects are produced mainly by the routing load, as hypotized in chapter 3.

As we could expect, the load increases in proportion to q; indeed, recalling the fundamental inequality

$$p_M \geq \frac{q-1}{q}$$

that represents the probability for a query to enter the hot zone, it's reasonable that, the greater is q, the greater is p_M.

In the Kademlia-like model, on the basis of the theoretical analysis we've carried out, a sensibly different constant was exptected to be hidden in the Ω-notation, with the respect to the previous model. Nevertheless, results seem to confute this hypotesis; tracing each query's lookup path in greater detail, we've seen that a particular reason can be adduced: often, when choosing a "less loaded" next hop in proximity of the target, a hot node was selected (recall that these nodes were kept available). Letting hot nodes go down as any other node, we've seen that sometimes the query failed in finding out its target (that is, cached values were not retrieved).

A final observation concerns routing tables' organization: we've noticed that, as the number of performed lookups grew up, the similarity found out among different nodes' routing tables grew up too. Curiously, this behavior is very similar to that described in [18, 8]: both works focus on an experimental evaluation of the Kad [3] network.

Authors emphasize how one of the most important problems exhibited by Kad is the *lookup inconsistency*: while an important reason can be identified with the phenomenon of churn, another reason, often misunderstood, is, as depicted in [18], the *similarity* characterizing routing tables of nodes located around a target ID.

The study reported in [18] monitors the Kad network using aMule clients. Whereas reporting many significant experimental results, this work is restricted to detect the high similarity exhibited by the routing tables of nodes close to a target node T: for example, authors notice that, comparing any pair of routing tables owned by nodes belonging to P (where P represents the set of nodes whose 16-bit long ID prefix matches the T's ID prefix), 70% of entries are identical. As a consequence, peers return similar and duplicated entries when a searching node queries them for T. In this way, although a GET lookup is able to find some of the closest nodes to a target, the STORE operation could have published binding

[3] The most popular implementation of the Kademlia overlay protocol; its main clients are eMule and MLDonkey.

information to some nodes really far from the target, since it failed in retrieving closer nodes.

Such a phenomenon clearly causes subsequent inconsistencies for the lookups performed by nodes. Since peers surrounding the target have similar "close-to-target" contacts in their routing tables, the same contacts are returned multiple times in response messages and the number of learned nodes is small. This duplicated contacts, sometimes accompanied by out-of date routing table entries (corresponding to nodes that have left the network), cause a Kad lookup to locate only a small number of the existing nodes close to the target, eventually failing in retrieving available content.

6. CONCLUSIONS AND OPEN PROBLEMS

At the end of our thesis, let us sum up the main contributions of this work and suggest its possible future developments.

The problem we've coped with is at the core of P2P overlays. A P2P network should scale to large and diverse node populations and provide adequate performance to serve all the requests coming from the end-users: nevertheless, often this goal is not achieved, since uneven request workloads may adversely affect specific nodes responsible for popular objects.

The majority of the DHT overlays proposed nowadays can't adequately handle this uneven workload: in this work, a very general DHT model has been qualitatively and quantitatively analyzed, in order to show that, in spite of any utilized caching mechanism, a malicious adversary can always reach the goal of "swamping" a restricted set of network's nodes with an unbearable amount of queries.

One of the main causes of this lack of load balancing has been identified with the absence of an appropriate relationship between caching mechanism and routing algorithm: as long as the latter is kept unaware of the value stored within nodes' caches, no caching strategy can provide effective results.

Therefore, we've tried to better exploit the potential of a tighter connection between routing and caching. In particular, in order to make this challenging problem more manageable, we've introduced a subtle distinction between two different types of load experienced by a node: the *request load* and the *routing load*.

On this basis, we've tried to answer a well defined question: "Is it possible to keep at most logarithmic the load experienced by any node, in any networking scenario"? Our study prospects an affirmative answer: two caching algorithms have been proposed, and their robustness has been analyzed against an adaptive adversarial scenario. The first strategy guarantees, even in the worst case, a logarithmic per-node load and a logarithmic lookup efficiency; nevertheless, it does not noticeably improve routing performances in normal networking conditions, as a caching scheme is expected to do. Therefore, we think it could be adopted if accompanied by a classic caching scheme (for instance, that proposed in [4]) and utilized in order to handle "emergency situations". The second algorithm provides a logarithmic per-node load and (intuitively) a *polylogarithmic* search performance. Nevertheless, it seems the more interesting between the two, and future work should focus on providing a complete, formal quantitative analysis.

In order to evaluate the performance of our protocol, it would be interesting to conduct a quantitative analysis, by means of experimental tests: in this way, the overall performance provided by the network could be judged in a real, and not only theoretical, context. Furthermore, the capability of our scheme to support segmented downloading could be evaluated as well: nowadays, multisource downloading is an emergent approach, that lets users download files in a more efficient way from many peers at once (the single file is downloaded, in parallel, from several distinct sources). Finally, it should be adequately studied how the lack of flexibility in neighbors selection, exhibited by our model, could affect the performance of the network: this aspects could produce negative effects (that is, latency increasing) mainly in widely deployed networking environments.

BIBLIOGRAPHY

[1] Abraham, I., Awerbuch, B., Azar, Y., Bartal, Y., Malkhi, D., and Pavlov, E. (2003). A generic scheme for building overlay networks in adversarial scenarios. In *Proceedings of the 17th International Parallel and Distributed Processing Symposium*, p. 40.2.

[2] Adler, M., Halperin, E., Karp, R.M., and Vazirani, V.V. (2003). A stochastic process on the hypercube with applications to peer-to-peer networks. In *Proceedings of the 35th annual ACM symposium on Theory of Computing*, pp. 575-584.

[3] Anceaume, E., Brasileiro, F., Ludinard, R., and Ravoaja, A. (2008). Peercube: an hypercube-based P2P overlay robust against collision and churn. In *Proceedings of the IEEE International Conference on Self Autonomous and Self Organising Systems (SAS).*

[4] Bianchi, S., Serbu, S., Felber, P., and Kropf, P. (2006). Adaptive load balancing for DHT lookups. In *Proceedings of the 15th International Conference on Computer Communications and Networks (ICCCN)*, pp. 411-418.

[5] Chen, J., Long, H., and Liang, L.(2008). Distributed hot spots caching mechanism for queries with popular distribution. *International Symposium on Computer Science and Computational Technology*, vol. 2, pp. 418-421.

[6] Cohen, E., and Shenker, S. (2002). Replication strategies in unstructured peer-to-peer networks. In *Proceedings of ACM SIGCOMM '02*, pp. 177-190.

[7] Dabek, F., Kaashoek, M.F., Karger, D., Morris, R., and Stoica, I. (2001). Wide-area cooperative storage with CFS. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, pp. 202-215.

[8] Falkner, J., Piatek, M., John, J., Krishnamurthy, A., and Anderson, T. (2007). Profiling a million user DHT. In *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement*, pp. 129-134.

[9] Ferreira, R.A., Jagannathan, S., Grama, A. (2006). Locality in structured peer-to-peer networks. Journal of Parallel and Distributed Computing, vol. 66, issue 2, pp. 257-273.

[10] Gopalakrishnan, V., Silaghi, B., Bhattacharjee, B., and Keleher, P. (2004). Adaptive replication in peer-to-peer systems. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS)*, pp. 360-369.

[11] Gu, Y., Chen, L., and Chen, L. (2009). A load balancing method under Zipf-like requests distribution in DHT-Based P2P Network systems. *2009 International Conference on Web Information Systems and Mining*, pp. 656-660.

[12] Guangmin, L. (2009). An improved Kademlia routing algorithm for P2P network. In *Proceedings of the 2009 International Conference on New Trends in Information and Service Science (NISS)*, pp. 63-66.

[13] Gummadi, K., Dunn, R., Saroiu, S., Gribble, S.D., Levy, H.M., and Zahorjan, J. (2003). Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. In *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, pp. 314-329.

[14] Gummadi, K., Gummadi, R., Gribble, S., Ratnasamy, S., Shenker, S., and Stoica,I. (2003). The impact of DHT routing geometry on resilience and proximity. In *Proceedings of the conference on Applications, Technologies, architectures, and protocols for computer communications (SIGCOMM)*, pp. 381-394.

[15] Gupta, A., Dinda, P., and Bustamante, F.E. (2005). Distributed popularity indices. In *Proceedings of ACM SIGCOMM.*

[16] Haray, F., Hayes, J.P., and Wu, H. (1988). A survey of the theory of hypercube graphs. *Comput. Math. Appl. 15*, pp- 277-289.

[17] Iyer, S., Rowstron, A., and Druschel, P. (2002). Squirrel: a decentralized peer-to-peer web cache. In *Proceedings of the 21st annual symposium on Principles of distributed computing*, pp. 213-222.

[18] Kang, H.J., Chan-Tin, E., Hopper, J., and Kim, Y. (2009). Why Kad lookup fails. Tech. report, University of Minnesota.

[19] Kangasharju, J., and Ross, K. W. (2001). Adaptive replication and replacement in P2P caching. In *Proceedings of the 6th International Workshop on Web Content Caching and Distribution.*

[20] Karger, D., Lehman, E., Leighton, T., Panigrahy, R., Levine, M., Lewin, D. (1997). Consistent Hashing and Random Trees: distributed caching protocols for relieving Hot Spots on the World Wide Web. In *Proceedings of the 29th annual ACM symposium on Theory of computing*, pp. 654-663.

[21] Leibowitz, N., Bergman, A., Ben-Sahul, R., and Shavit, A. (2002). Are file swapping networks cacheable? Characterizing P2P traffic. In *Proceedings of the 7th International Workshop on Web Content Caching and Distribution.*

[22] Loguinov, D., Kumar, A., Rai, V., and Ganesh, S. (2003). Graph-theoretic analysis of structured peer-to-peer systems: routing distances and fault resilience. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pp. 395-406.

[23] Lv, Q., Cao, P., Cohen, E., Li, K., and Shenker, S. (2002). Search and replication in unstructured peer-to-peer networks. In *Proceedings of the 16th ACM International Conference on Supercomputing (ICS'02).*

[24] Maymounkov, P., and Mazieres, D. (2002). Kademlia: a peer-to-peer information system based on the Xor metric. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS'02).*

[25] Mitzenmacher, M., and Upfal, E. (2005). Probability and computing: randomized algorithms and probabilistic analysis. Cambridge University Press.

[26] Plaxton, C. G., Rajaraman, R., and Richa, A. W. (1997). Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the 9th annual ACM symposium on Parallel algorithms and architectures*, pp. 311-320.

[27] Ramasubramanian, V., and Sirer, E.G. (2004). Beehive: O(1) lookup performance for power-law query distributions in peer-to-peer overlays. In *Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, p. 8-8.

[28] Rao, W., Chen, L., Fu, A., and Bu, Y. (2007). Optimal proactive caching in peer-to-peer Network: analysis and application. In *CIKM '07: Proceedings of the 16th ACM Conference on information and knowledge management*, pp. 663-672.

[29] Rao, A., Lakshminarayanan, K., Surana, S., Karp, R., and Stoica, I.(2003). Load balancing in structured P2P systems. In *Proceedings of the International Workshop Peer-to-Peer Systems (IPTPS)*, pp. 119-128.

[30] Ratnasamy, S., Francis, P., Handley, M., Karp, R., and Shenker, S. (2001). A Scalable Content-Addressable Network. In *Proceedings of the ACM SIGCOMM '01*, pp. 161-172.

[31] Risson, J., and Moors, T. (2004). Survey of research towards robust peer-to-peer networks: Search methods. Tech. repoty UNSW-EE-P2P-1-1, University of New South Wales, Sydney, Australia.

[32] Rodrigues, R., and Liskov, B. (2005). High availability in DHTs: erasure coding vs. replication. In *Proceedings of the 4th International Workshop on Peer-to-Peer Systems.*

[33] Rowstron, A., and Druschel, P. (2001). Storage management and caching in PAST, a large-scale persistent peer-to-peer storage utility. In *Proceedings of the ACM SOSP '01.*

[34] Rowstron, A., and Druschel, P. (2001). Pastry: scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms*, pp. 329-350.

[35] Saad, Y., and Schultz, M. (1988). Topological properties of hypercubes. *IEEE Transactions on Computers, 37(7).*

[36] Saleh, O., and Hefeeda, M. (2006). Modeling and caching of peer-to-peer traffic. In *Proceedings of the 2006 IEEE International Conference on Network Protocols*, pp. 249-258.

[37] Schlosser, M., Sintek, M., Decker, S., and Nejdl. W. (2002). HyperCuP - Hypercubes, Ontologies and Efficient Search on P2P Networks. *International Workshop on Agents and Peer-to-Peer Computing*, Bologna, Italy.

[38] Shen, H. (2008). EAD: an efficient and adaptive decentralized file replication algorithm in P2P file sharing systems. *Eighth International Conference on Peer-to-Peer computing*, pp. 99-108.

[39] Stading, T., Maniatis, P., and Baker, M. (2002). Peer-to-peer caching schemes to address flash crowds. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems.*

[40] Stoica, I., Liben-Nowell, D., Morris, R., Karger, D., Dabek, F., Kaashoek, M. F., and Balakrishnan, H. (2001). Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM*, pp. 149-160.

[41] Tewari, S., and Kleinrock, L. (2006). Proportional replication in peer-to-peer networks. In *Proceedings of the 25th IEEE International Conference on Computer Communications*, pp. 1-12.