Improving the Dependability of Prefix-Based Routing in DHTs

Sabina Serbu, Peter Kropf and Pascal Felber

University of Neuchâtel, CH-2009, Neuchâtel, Switzerland
{sabina.serbu, peter.kropf, pascal.felber}@unine.ch

Abstract. Under frequent node arrival and departure (churn) in an overlay network structure, the problem of preserving accessibility is addressed by maintaining valid entries in the routing tables towards nodes that are alive. However, if the system fails to replace the entries of dead nodes with entries of live nodes in the routing tables soon enough, requests may fail. In such cases, mechanisms to route around failures are required to increase the tolerance to node failures.
Existing Distributed Hash Tables (DHTs) overlays include extensions to provide fault tolerance when looking up keys, however, these are often insufficient. We analyze the case of greedy routing, often preferred for its simplicity, but with limited dependability even when extensions are applied.
The main idea is that fault tolerance aspects need to be dealt with already at design time of the overlay. We thus propose a simple overlay that offers support for alternative paths, and we create a routing strategy which takes advantage of all these paths to route the requests, while keeping maintenance cost low. Experimental evaluation demonstrates that our approach provides an excellent resilience to failures.

Key words: fault tolerance, reliability, DHT, routing.

1 Introduction

Dependability concerns many properties of a system, such as scalability, reliability, security, data integrity, availability, routing or fault tolerance. These properties are generally dealt with according to the system's architecture. In this paper we address the *accessibility* of the stored data, focusing on the routing and fault tolerance issues in structured P2P systems.

Consistent information about the nodes in the system is crucial for effective operation and it is essential that a peer-to-peer system allows for fault tolerance. Thus, a silent node departure shall not turn the whole or part of the system inoperable. Though information (data) may disappear from the system when a node silently departs, the routing functions that it assumed shall be taken over by other peers alive. This calls for some system management and maintenance functions as far as the overlay network organization and associated routing functions are concerned. Clearly there is a trade-off between the costs of such maintenance and the effectiveness of the routing achieved, which depends of course on the properties of the particular overlay structure.

The main trend in existing P2P systems is to essentially focus on preserving stored data and preserving consistency of the network structure, ignoring *fault-tolerant access* to the data under churn, i.e., frequent node arrival and departure. To provide data accessibility in a system under churn, both a fault-tolerant infrastructure and fault-tolerant routing need to be taken into account. In the following, we present these aspects and later in Section 2 we show some of the existing methods providing for fault tolerance.

Distributed hash-tables (DHTs) use specialized placement algorithms to assign responsibility for each object to peers as well as "directed search" protocols to efficiently locate objects. With regard to the infrastructure aspect, they rely on a large variety of different structures, such as rings, multidimensional spaces, hypercubes or other types of graphs. One notable difference between these structures is the *degree* that each node in the overlay has, which, in this context, is the number of neighbors with which a node maintains continuous contact for supporting the routing mechanism. A *constant node degree* assures low maintenance costs for the entries in the routing tables, costs related to the control traffic that is required to check for the state of the neighbors and to set a new node for an entry that is found to contain a dead node. Unfortunately, this also means that they do not offer a significant tolerance to faults. Examples include de Bruijn-based overlays [1], Viceroy [2] or CAN [3]. Other DHTs use a *logarithmic node degree*, such as Chord [4], Pastry [5], Tapestry [6] or Kademlia [7]. These systems show higher costs for maintaining the routing tables compared to the systems that use a constant node degree. Nevertheless, they can use alternative entries when an entry fails, which provides a good start base towards a *fault-tolerant infrastructure*. This is the reason why, in our research, we are focusing on this type of overlays.

An overlay infrastructure needs to be able to recover from failures by replacing entries of dead nodes with entries of live nodes in the routing tables. To update such an entry in the routing table, one must find a node that would fit at that entry. Since it is costly and mostly impossible to keep all routing tables entries always populated with live nodes, these updates are made periodically: at each time interval, maintenance requests are issued and the routing tables are updated. As a consequence, this still leaves a time window when entries may refer to dead nodes. Because routing table entries become often invalid under churn, the system has to additionally provide *fault-tolerant routing* by finding alternative routes to forward the requests towards the destination. As in [8], we say that the overlay routing is *dependable* if a request reaches its destination.

To discuss fault-tolerant routing, we illustrate in Section 3 the case of Chord-like DHTs which use *greedy routing*, one of the most-known and widely-used routing algorithm. Greedy routing is simple: at each routing step, the request is directed towards a node as close as possible to the destination. This strategy provides fast lookup because the number of hops is minimized. In case of node failure, greedy routing algorithms typically apply a "route around" strategy by using a lower entry from the routing table if the normally chosen entry contains a dead node. Experimental results have confirmed that greedy routing under node failures is an unreliable strategy with respect to fault tolerance and routing dependability. Indeed, the advantage of getting as close as possible to the destination at each routing step (i.e., going as far as possible from the source) becomes a disadvantage under node failures, as this strategy exploits only a small part of the possible paths. At each routing step, the number of possible paths towards desti-

nation is heavily decreasing, which drastically diminishes the chances of finding a valid path to destination.

Following the analysis of standard greedy routing, we propose an overlay structure and a routing scheme to provide a high degree of fault tolerance, while still keeping maintenance costs low:

- the *hypeer* overlay is a logarithmic node degree DHT with a structure that approximates a hypercube.
- the FT-routing strategy is an efficient routing scheme that allows multiple options in the selection of a next node in the request path to provide fault tolerance in case of churn.

The *hypeer* overlay offers the choice between many redundant paths, which is needed in a fault tolerant system. As all other DHTs, it uses an identifier space where the nodes and the keys obtain IDs in a form of a sequence of binary digits. To route towards a node responsible for the requested key, several intermediate nodes are traversed such that the digits from the source identifier are successively replaced by the digits of the key identifier. Our proposed structure is loosely based on a hypercube. This offers the possibility of treating the digits in any order when routing from source to destination, which tunes the number of redundant paths. The redundant paths considerably enhance fault tolerance.

The rest of the paper is organized as follows. In Section 2 we discuss related work on hypercubes and fault-tolerance support. Section 3 further details the motivation of our work. In Section 4 we present our system: its structure, routing strategies and functionality. In Section 5 we present the experiments conducted and discuss the results obtained. Then, we conclude in Section 6.

2 Related Work

In this section we present related work for hypercubes, which represents the structure that offers the highest choice for alternative paths, and then some of the existing solutions for fault-tolerance with respect to the infrastructure and the routing strategies. Note that we do not deal with any security aspects, such as trusted nodes or trusted information (this is well detailed in [9]).

2.1 Hypercube-based DHTs

There are several DHTs that use the hypercube structure in order to provide alternative paths. This allows for fault tolerance, however with a penalty of increasing the complexity of the overlay maintenance.

The eQuus[10] system has a topology of a partial hypercube. Each vertex represents a clique, i.e., a group of nodes that are close in terms of a proximity metric. The lookup procedure is similar to Pastry[5], with the main difference that each entry represents a clique and not a single node. The nodes in a clique share the same ID and keys. Thus, fault tolerance is mostly treated from the point of view of data availability, and not to achieve routing fault tolerance.

Schlosser *et al.* [11] present HyperCuP, a hypercube structure that is built as peers join the system. A node keeps its neighbors on a per-dimension basis, and it might have the same node as neighbor in two or more dimensions, if no other suitable node has been found. When joining the system, a new node contacts an existing random node which will become its new neighbor in a dimension of its choice. The strongest point of this solution is the idea of the hypercube construction, however a node may become responsible of too many vertices of the hypercube, thus its failure may severely affect the routing. Moreover, the usage of broadcast messages to all or a part of the dimensions of the hypercube may become too costly in terms of number of messages.

In [12], Alvarez *et al.* propose to increase the number of path connections through the use of a hypercube structure. Each node has an identifier and a mask that indicates the ID space that the node is responsible for. The routing algorithm can be either proactive, assuring a specific route to each node based on a tree distribution of the IDs, or reactive by creating on demand a route and keep it for a certain period of time. The usage of route creation makes this solution seem more adequate for systems where churn rates are rather low.

2.2 Fault Tolerance with other Structures

In order to achieve fault tolerance, the resource discovery mechanism described in [13] is based on an arrangement of multiple Chord rings, each one responsible for a keyword. However, the system relays on a super ring which contains pointers to each Chord ring. This solution aims for fault tolerance, however the super ring is a critical point of failure.

Wepiwé *et al.* [14] propose a concentric multi-ring overlay for high reliability, where the nodes on a given inner ring form a de Bruijn graph. This overlay assumes knowledge about the reliability of the nodes, which normally is not a constant in any system.

2.3 Extensions for Fault-Tolerance

Backup Nodes. The easiest and most widely adopted solution to deal with dead nodes in routing tables is the addition of *backup nodes* (redundant links). The best-known examples are systems like Chord [4] or Pastry [5]. In Chord, each node maintains a list of a fixed number of successors on the ring. When an entry has failed, a lower entry is used. For the lowest entries, the list of successors may be used. Lam *et al.* [15] propose the K-consistent networks. Each node keeps always K nodes at each entry in its routing table. Whenever a node from the routing table fails to respond, a repair mechanism trie to find a new suitable node for the same entry. This type of solutions is obviously limited by the number of backup nodes used. A high number of backup nodes means a higher number of alternative paths, and so a higher probability of success. However, the backup nodes need also maintenance, so the disadvantage is seen in the additional costs imposed by maintaining more node entries in the routing tables.

Reducing the number of dead entries. Castro *et al.* [8] use a different approach to the ones mentioned so far, proposing techniques to detect node failures and repair routes. They apply this solution in MSPastry, a particular implementation of Pastry. These techniques successfully decrease the number of dead entries in the routing tables, however

there is no solution to completely eliminate them, which means that there is still the need for routing around failures.

Replication. Replication is one of the most simple solutions for fault-tolerance, where several replicas of the same object are placed at different nodes. These nodes are either chosen uniformly in the identifier space, in the neighborhood of the destination, or using a replica function [9]. Replication can be easily applied as a complementary solution to any fault-tolerant infrastructure or routing solution.

2.4 Fault-Tolerant Routing

For dependable routing under failures, Aspnes *et al.* [16, 17] propose two extensions for greedy routing. When a node cannot find another node that is closer to the destination than itself, it can use either *random re-route* (random choice of another node to forward the request to), or *backtracking* (sending back the request to the previous node in the request path by keeping track of some visited nodes). These two extensions provide reasonable results, however, they still exploit only a small number of possible paths.

Backtracking is however a good technique to enlarge the number of alternative paths, but it is not well exploited when used with greedy routing. A request that gets close to the destination, but is forced to use backtracking, would do small back hops, which means that the gained number of alternative paths remains small.

Another possibility to increase the request success rate is *redundant routing* (as it is called in [9], or *parallel routing* as in [18]). In this case, several copies of the same request are sent towards the same destination through different paths. In [9], for fault-tolerance, such copies are sent from the source to a set of its neighbors towards the nodes that own replicas of the requested object and following different paths. Independently of the choice for the next hops of the paths, when applying redundant routing, more requests are sent in the system, so more processing is required at the nodes. This, obviously, increases the costs considerably.

In contrast to the existing solutions for fault tolerance, we aim to improve *dependability* by allowing at each routing step to consider the maximum number of possible alternative paths, even if no failure has been detected yet.

3 Motivation

To provide a high level of fault tolerance, we consider it necessary to take into account both the overlay and the routing strategy.

3.1 On The Dependability of Greedy Routing

Many DHTs use greedy routing to forward the requests because of its simplicity. This strategy gives good results in terms of path length, but it has limitations in the number of paths it can exploit. To get from source to destination, greedy routing adjusts the bits from left to right when the request is forwarded to the next hop in the request path. This strategy generally leads to *path convergence*: the last hops of most requests for a certain

destination pass through only a small set of nodes, which are mostly the preceding neighbors. Under a failure-free operation, these nodes are likely to be overloaded if the destination is very popular. Furthermore, if one of them fails, the traffic will be severely affected.

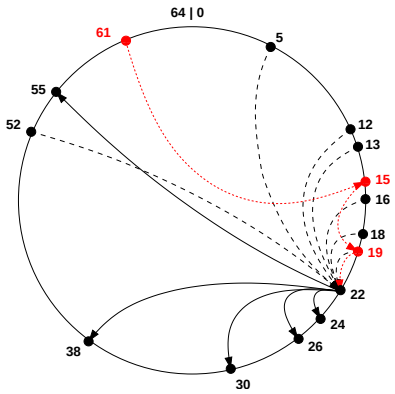

Fig. 1. Example of Chord with an identifier space of $2^m = 64$.

Systems such as Chord [4] or Pastry [5] suffer from these limitations of greedy routing. A graphical representation of a Chord example is shown in Figure 1. In Chord, each node and object has a m-bit identifier on a 2^m ring, obtained by respectively hashing the IP address and the name. The objects are mapped to their subsequent node on the ring. For routing purposes, each node has a routing table with m entries, each entry i pointing towards the first node on the ring at a distance of minimum 2^i, where $i = 0..m-1$. Conversely, each node is in the routing table of other nodes, so it has incoming links from these nodes. In the example of Figure 1, 15 (out of 40) nodes are shown on a 2^6 ring. The incoming links of node 22 are shown with dashed dark lines, and its outgoing links are shown with solid dark lines. While each of the outgoing links points to nodes at a distance close to a power of 2 away, the distance from the incoming link nodes is less predictable. Each request is forwarded by greedy routing, following always a clockwise path, as for example the request going from node 61 to node 22 in three hops (the dashed grey line).

Figure 2 shows, in percentages, the cumulative distribution function (CDF) of the number of requests that are received per incoming link in a Chord-like system with no node failures. In this experiment, the identifiers are mapped on $m = 15$ bits. The system has 10,000 nodes and 20,000 keys, with 200,000 requests uniformly issued. As can be seen on the left-hand side of the graph, most of the traffic is received from the incoming links with small distances, which limits the possibilty for redundant paths. More than 80% of the traffic is received from the incoming links at 2^i away, where $i \leq 4$. Note that the first entries of the routing table (small values of i) may point to the same node because of the inter-node distance. Thus, the predecessors of a node are critical nodes because they bear the majority of the traffic for that node. If such a node fails, the rate of request success drastically decreases.

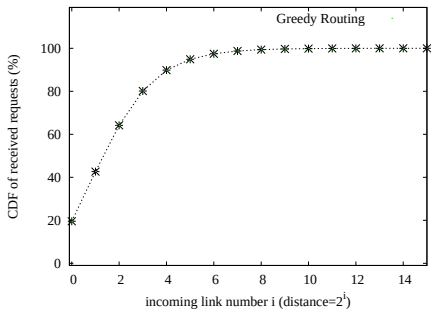

Fig. 2. The Cumulative Distribution Function (CDF) of the percentage of received requests per incoming link (2^i) in a Chord system using greedy routing.

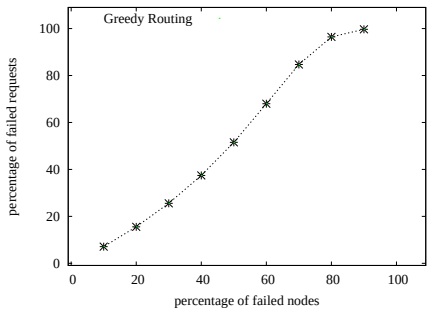

Fig. 3. Failure rate of greedy routing. The high percent of failures shows that greedy routing does not exploit the redundant paths.

In the experiment under failures, whenever a routing table entry refers to a failed node, a lower entry is used instead. Figure 3 shows the percentage of failed requests when varying the percentage of failed nodes. The graph shows that this strategy is not dependable: when half of the nodes fail, half of the requests also fail.

The main reason for this poor performance is that alternative paths are not exploited. At each node, the request is sent as close as possible to the destination. This means that the distance between a next hop node n_{nh} and the destination n_d of a request is minimal. Unfortunately, it also means that the number of possible alternative paths is minimal once the request has reached n_{nh}. As an example, a request in Figure 1 goes from node $n_s = 61$ to node $n_d = 22$, by going through nodes 15 and then 19, according to greedy routing. The possible alternative paths go through each of the incoming links of n_d: 5, 12, 13, 16, 18 and 19. At n_s, greedy routing chooses $n_{nh} = 15$. When this node is reached, the request (which always travels clockwise) may pass only through the incoming links 16, 18 and 19. If these nodes fail, the request will also fail to reach its destination, even though alternative paths from n_s through the incoming links of n_d, nodes 5, 12, or 13, would be valid.

Another aspect of this kind of overlays is that the outgoing links of a node are not exactly at 2^i distances, so the request does not necessarily follow 2^i jumps. This fact prevents from applying a deterministic routing strategy to exploit other valid paths.

All these observations uncover the mismatch between the goal of providing fault tolerance and the means used for lookup with greedy routing. The extensions for fault-tolerance may give good results, however, if better support for exploiting alternative paths is already considered at overlay design time, even better results may be obtained.

3.2 The Hypercube Structure Approach

Based on the above observations, we seek for a structure that best provides alternative paths, where the routing algorithm is able to process in any order the digits from the source identifier to match that of the destination. In such cases, the total number of

available paths is $m!$, where m is the number of digits in the identifier sequence. After i bits have been treated, the number of available paths is $(m-i)!$. Such a routing procedure that exploits alternative paths is well achievable in *hypercube structures*. Some hypercube based systems were already described in Section 2.

In a hypercube architecture with N nodes, each node has a $O(\log N)$ node degree, where the number of neighbors is equal to the number of dimensions. Each neighbor has an identifier that differs by exactly one digit.

The problem of using the hypercube for an overlay is that it requires complex protocols to deal with churn. When new peers join, the number of dimensions has to be increased. Conversely, when peers leave the system, the overlay has to treat the dimension split problem. Because of these problems and the high costs their treatment induces, deterministically constructed hypercubes are not suitable to provide fault tolerance under high churn.

However, to take advantage of the (structural) availability of alternative paths for routing in a hypercube, while avoiding the drawbacks of increased maintenance costs for rearranging the structure in case of churn, one could use an approximation of a hypercube that is built probabilistically.

In our study, we seek to achieve this by assigning non-random IDs when peers join. The idea is to assign to a new node n_b an ID that is exactly at a power of 2 away from an existing random node n_a. Thus, a hypercube vertex is set to n_b and an edge is created from n_a to n_b. As more peers join, new dimensions of the hypercube fork spontaneously by populating its edges and vertexes. In terms of the overlay structure, after node n_b joins the system, node n_a will have n_b at entry i of its routing table, and n_b will be at exactly 2^i away from n_a.

Figure 5 illustrates this structure, which we call a pseudo-hypercube. It has an almost deterministic node placement, and as a consequence, we can route almost deterministically. Greedy routing is still supported, but we can also use non-greedy routing algorithms that are more efficient for fault tolerance, as presented in the next section.

4 *hypeer* Design

Our design goal is based on the discussion in the previous sections. We propose *hypeer*, a DHT with a ring structure embedded in a directed pseudo-hypercube supporting several routing strategies.

The nodes and the keys have IDs in an identifier space of length 2^m, where m is the number of bits in the identifier sequence. The responsibility for a key is given to the first node that follows the key on the ring (as it is the case in Chord).

In contrast to the common method of using a hash function to map the nodes on the ring, we choose to assign the node identifiers in a way to approximate a hypercube structure by trying to maintain an even inter-node distance equal to a power of 2 despite churn. We call *inter-node distance* the distance between a node and its predecessor.

Each node has a link to its predecessor and successor on the ring (see Figure 4). The routing table of a node contains entries which are, with a high probability, at exactly 2^i away, with i from 0 to $m-1$. These are the neighbors of the node in the hypercube. The links between a node and its neighbors are shown with arrows in Figure 5. The

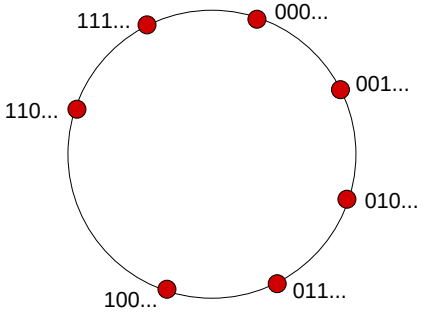

Fig. 4. The ring structure.

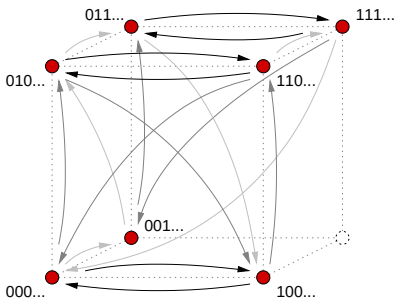

Fig. 5. The Hypercube structure.

nodes 2^{m-1} away from each other have bidirectional links (they are in "diametrical opposition" on the ring) and are denoted by black arrows. The gray arrows are the other links in the hypercube approximation.

The routing procedure is based on replacing the bits from the source ID by the bits of the destination ID, i.e., bit i from the identifier sequence is replaced through a jump to a node at 2^i away. Our hypercube structure provides in most cases links towards nodes that are exactly at 2^i away. However, in some cases, and mostly for small values of i, links may point to nodes that are not exactly at 2^i away. Thus, a jump to such a node would affect also some of the bits at the right-hand side of the replaced bit. One or more of the bits on the left side may be affected only when changing a bit from 1 to 0, because of the resulting carry over. Thus, in *hypeer* there are not only links towards nodes with only one bit changed as in a proper hypercube. To construct a proper hypercube, the i^{th} link should point counterclockwise if the i^{th} bit is set, and clockwise if it is not set, which would create double links between nodes.

A fault tolerant routing strategy must provide a large number of alternative paths even when the request is close to the destination (*close* means a small number of digits that are different in the source and destination identification sequences). As a consequence, we propose a solution where the request follows small steps in the beginning of the routing path (where the low-order digits are treated) and then longer steps (treating higher-order digits). It is clear that when no failures occur, the strategy gives similar results to greedy routing, because the number of hops to route a request is of the order $O(\log N)$, assuming that we can treat the bits in any order. However, when failures do occur, this strategy exploits a much higher number of possible paths, as the requests are routed around failures with a higher probability.

In short, we are applying simple modifications to Chord-like systems. Chord cannot easily exploit redundant paths because of its non-determinism in node placement that does not permit treating digits in any order, so we are fixing this by adding some determinism in the placement of the nodes. This is obviously advantageous for the routing strategy due to the control of node position. Besides fault tolerant routing, this structure can also adopt a routing strategy to balance the traffic load on the path towards a popular destination.

4.1 *hypeer* Overlay

Our system has the IDs assigned deterministically, with the purpose of creating a hypercube-like structure. The main idea is to maintain 2^i links between the nodes and implicitly an inter-node distance of a power of 2 in order to take advantage of the redundant paths of the hypercube for routing.

We start dealing with fault tolerance at design time of our overlay. As explained before, we do not use a hash function to map the nodes on the ring. When a new node arrives in the system, it sends a request to a random key (e.g., obtained by hashing the IP address of the new node). The node responsible for that key adds the new node as a 2^i neighbor. A new vertex and the edge between the existing node and the new node are thus populated. The joining procedure is shown in Algorithm 1.

Algorithm 1 Node n_a receives a Join Request

```
0: {First decide whether n_a or its predecessor assigns the ID}
1: if dist(pred(n_a), n_a) > dist(n_a, succ(n_a)) then
2:     Send the request to pred(n_a)
3: else
4:     {Check each entry x of the routing table RT, starting from the highest entry}
5:     entry ← x, where RT(x) ≠ n_a + 2^x
6:     if entry found then
7:         Respond with n_b ← n_a + 2^x
8:         RT(x) ← n_a + 2^x
9:     else
10:        Send request to pred(n_a)
11:    end if
12: end if
```

A new node n_b that wants to join the system has to issue a join request towards a random ID. The request is routed in the system until it arrives at node n_a which is responsible for that ID.

Node n_a consults its routing table to find the entry that has not yet a node that is exactly at a power of 2 away, by starting with the highest entries pointing to the furthest away nodes (lines 4-5). We thus give priority for perfect 2^i edges between nodes which are at long distances from one another on the ring (high i), because these are not affected by the inter-node distance that can vary. Node n_b will be assigned with an ID equal to (n_a + 2^x), where x is the entry found (lines 7-8).

In the case that n_a has at each entry x a node at exactly 2^x away, the joining request is sent to its predecessor (lines 9-10).

Node n_a is chosen randomly, as the associated ID was chosen randomly. As an optimization, with the objective to not partition the ID space into small pieces, n_a checks if its predecessor $pred(n_a)$ is further away than its successor $succ(n_a)$, and in such a case, it asks $pred(n_a)$ to assign the ID (lines 0-2). Of course, more complicated schemes to choose this node may be used for a better approximation of the hypercube. However, randomness gives good results, as we show later in Section 5. Besides creating the hypercube-like structure, this scheme also assures the uniformity of the distribution of the node IDs in the identifier space.

After n_b obtains its ID, it will start populating its routing table. Node n_b issues requests towards IDs that are 2^i away from itself, where i goes from 0 to $m-1$. Node n_a

will wait for a short period of time before updating its routing table with n_b to allow n_b to create its own routing table.

To detect node joins and departures, the routing tables are periodically updated, in the classic way of issuing requests to the ID that each entry should accommodate, i.e, a node n sends requests to all $n + 2^i$, with i from m to 0.

4.2 Routing Strategies

First we present our fault tolerant routing strategy that may be used with the *hypeer* overlay, and then we discuss other routing strategies, pursuing other goals such as traffic load balancing.

Fault-Tolerant Routing (FT-routing)

Algorithm. For fault tolerance, we want to take advantage of all available incoming links of a destination, starting from the source node and going clockwise towards the destination on the ring.

The idea of the routing algorithm is to reach the nodes that have direct 2^i links towards the destination. These nodes are, with a high probability, the nodes that are responsible for the IDs $keyId - 2^i$, where $keyId$ is the requested key and i goes from 0 to $t - 1$. The value of t depends on the distance between the source node and the key, where $keyId - 2^t$ represents the furthest incoming link of the destination that is between the source and the destination on the ring.

The routing algorithm at node n_x is presented in Algorithm 2. We show later in this subsection how we deal with node failures.

Algorithm 2 FT routing algorithm

```
1: upon receive lookup(T, key) at node n_x
2: {Receive request}
3: if pred(n_x) < key ≤ n_x then
4:     {Node n_x is responsible for key: success}
5: else if T ≤ pred(n_x) < key then
6:     {Went too far: send to the predecessor}
7:     Send lookup(T, key) to pred(n_x)
8: else
9:     {Compute next hop}
10:    FT-route(key, key)
11: end if
12:
13: function FT-route(T, key)
14: if ∃n_k ∈ RT s.t. n_k − range < T ≤ n_k then
15:    {This node is probably responsible for T}
16:    Send lookup(T, key) to n_k
17: else
18:    {Compute new T}
19:    T ← T − 2^i, where i is max in T − 2^i > n_x
20:    FT-route(T, key)
21: end if
```

Upon receiving a lookup request, node n_x directly responds to the request if it is responsible for the key (lines 1-4) and the request path ends here.

At each node n_x from the request path, a next node on the request path towards the requested key needs to be found. We use a recursive function, which we call FT-route(T, key), which has the parameters T and key, and which terminates when it has found a node to forward the request to. T is a target (an identifier) that we aim to reach on the ring from n_x in a single hop. If n_x is not able to directly send the request to T, a new target T is set and the function is called recursively. In the first call of FT-route(T, key) at any node, T is set to the destination key, and then T decreases in the subsequent recursive calls with the largest power of 2 possible such that T is still between the source and the destination on the ring. With each call, the power of 2 decreases and the target gets closer to the source.

In the following, we say that a node is responsible for an identifier, if the identifier is between the node and its predecessor on the ring (in the same way as we map keys to nodes). Furthermore, we say that a node is *probably* responsible for an identifier when the identifier is between the node and an estimation of the position of its predecessor on the ring. When we are not aware of the position of the predecessor (as it is the case for the predecessors of the nodes in the routing table) we use an estimation of the inter-node distance.

Node n_x needs to find a node n_k from its routing table which is probably responsible for the target T. The condition at line 14 is satisfied, if T lies on the ring between $n_k - range$ (the assumed predecessor node of n_k, where $range$ is the estimation of the inter-node distance) and n_k. Since the nodes are uniformly distributed on the ring and because a large enough estimation of the inter-node distance shall be used, we set $range$ as the maximum between the inter-node distance of n_x and the inter-node distance of its successor. This can be easily computed since n_x knows its predecessor and successor IDs. Other values might be used as well, for example the average of the inter-node distance of all the nodes that the request passed through, which of course would add this value to the request message.

The chosen estimation works well in most cases due to the fact that the inter-node distance is the same for the majority of the nodes, as we will show later in Section 5. However, in some cases it happens that the request is sent further away than the node responsible for the target. The request is then sent to the responsible node by using one or more (but only few) hops through predecessor links (lines 5-7). To treat such cases, the target itself is added to the request message when forwarding the request to a probably responsible node for a target (line 16).

If n_x did not find a suitable node in its routing table to forward the request towards T, it will set a new target, which is 2^i before the current target on the ring. For the fault tolerance goal, we choose i as the maximum value between 0 and $m - 1$ such that the new target is still after n_x on the ring (lines 18-20). This assignment for recursive calls creates a virtual path where each hop is a power of 2, and moreover, the powers of 2 increase with each hop.

Note that the algorithm provides flexibility for the order of fixing the digits. If we choose i as the minimum value, this leads to a form of greedy routing. Moreover, if i is chosen randomly, this leads to a routing strategy that balances the load on the incoming links of the destination. This latter strategy is further detailed in the next subsection.

Treating Failures. In the following, we show how the above algorithm deals with the failures of nodes in the system.

If no entry from the routing table of n_x points to a live node (from those who point toward nodes before the destination on the ring), we say that a dead-end has been reached. Then, the request is backtracked to the previous node in the request path. The same happens when the request needs to be sent to the predecessor (line 5 of Algorithm 2) and the predecessor is down or a dead-end. To be able to use backtracking, before forwarding a request, a node adds its ID to the request message.

However, if the suitable entry is down or a dead-end, but other nodes from the routing table are not, n_x chooses another entry as follows. If n_x has found a node that is probably responsible for the target, but it is down or a dead-end, it will choose a node at a higher entry (starting from the following entry, going up), in order to "jump" over that target. If no node is found suitable from the higher entries (i.e., alive, not a dead-end and before the destination on the ring), a smaller entry (starting from the previous entry, and decreasing) is used. We choose first the higher entries and then the smaller ones because we want the request to quickly by-pass the faulty target, and not to increase unnecessarily the path length in the attempt to reach the same target that seems to belong to a dead node.

Without loss of generality, we do not consider failures of the destination node, as we treat fault tolerance from the routing point of view. If the destination node is down, its keys are lost anyway. Any request for such a key would then result in a non successful lookup.

Other Routing Strategies

LB-routing and GR-routing. Alternative routing strategies to the FT-routing algorithm described above include the following:

- *random-order routing (LB-routing)*, where the digits are replaced in a random order, but not necessarily independently. The fault tolerance is not as high as for the FT-routing, but considers traffic load balancing on the incoming links.
- *greedy routing (GR-routing)*, where the digits are treated from left to right. Our overlay supports also greedy routing, however, neither fault tolerance nor load balancing are expected.

Besides fault-tolerance, another advantage of alternative paths is to reduce the traffic load of the last nodes on the paths to a popular key. This is the reason for including LB-routing as a routing strategy in *hypeer*. LB-routing implements the same algorithm as FT-routing (presented in Algorithm 2), with the difference that at line 19, the new target T is chosen randomly. However, to maintain the same virtual path at each node, the targets have to be chosen the same at each node. Otherwise, setting different targets at each hop might increase significantly the path length.

The results for LB-routing have also been included in the experiments presented in Section 5 as a compromise between GR-routing and FT-routing.

To additionally improve fault tolerance, several copies of the same request may be forwarded using a different routing strategy. This technique is called redundant routing[9] or parallel routing[18].

5 Evaluation

In this section we analyze *hypeer*, a structure with an uniform partitioned space, and FT-routing in *hypeer*, as a good routing strategy with or without failures. To present the results, we compare FT-routing with GR-routing and LB-routing.

5.1 Overlay Structure

Our approach for assigning node IDs to new nodes ensures that they are at a distance of 2^i from some existing node. Further, on expectation, nodes are uniformly distributed in the identifier space. To validate this claim, we have simulated 10,000 node arrivals in an identifier space of 2^m, $m = 20$ and then computed the distribution of inter-node distances.

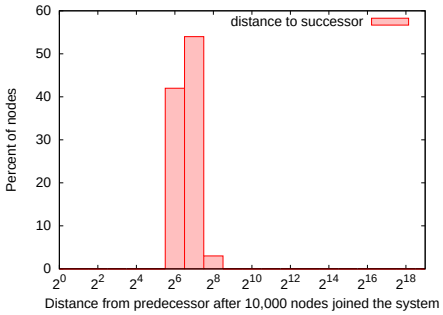

Fig. 6. Inter-node distance for 10,000 nodes in an identifier space of 2^{20}.

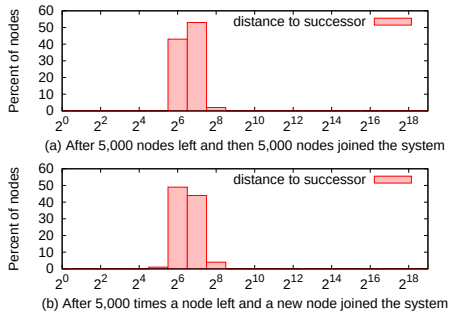

Fig. 7. Inter-node distance under churn.

As can be seen in Figure 6, among all 20 possible distance values, most of the nodes (roughly 95%) have an inter-node distance of either 2^6 or 2^7. Having two values with consecutive exponent for inter-node distance is expected, because of the continuous change in the hypercube structure caused by the new nodes arrival.

With this overlay we further analyzed the inter-node distance when dealing with churn. The results of two scenarios are depicted in Figure 7: (a) a scenario where 5,000 nodes leave and then 5,000 nodes join, and alternatively (b) a scenario with 5,000 successions of a leave followed by a join. Here, we observe that churn is only slightly affecting the overlay: the inter-node distances remain almost the same.

Next, we analyzed the number of outgoing links that are at exactly 2^i away from the current node, as shown in Figure 8. The maximum number of different outgoing links is 14 (out of m=20), because at least the first 6 entries of the routing tables point to the successor node, as a direct consequence of the inter-node distance of minimally 2^6. This explains the 0-percentage of nodes having a number of outgoing links larger or equal to 15 (the right-hand side of the graph), and also the smaller percentage for 14 outgoing links. The main observation is that the graph has an increasing tendance: a higher percentage of the nodes have a higher number of outgoing links towards nodes

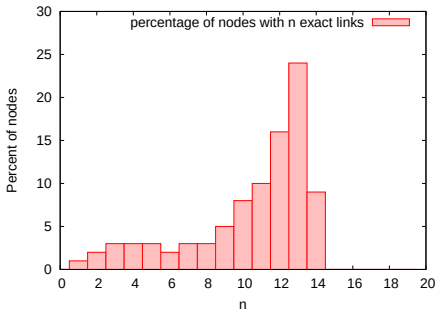

Fig. 8. Percent of nodes with the same number of outgoing links at exactly powers of 2 away, for 10,000 nodes in an identifier space of 2^{20}.

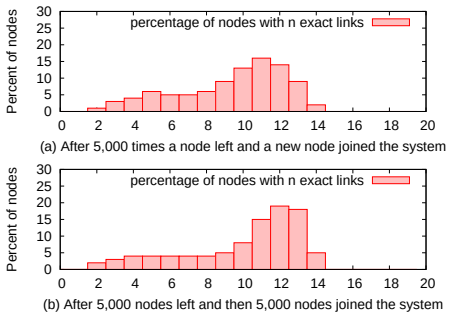

Fig. 9. Percent of nodes with the same number of outgoing links at exactly powers of 2 away under churn.

at 2^i away. This means that the hypercube edges at 2^{m-1} are populated first, and then the lower ones. The same observation holds when analysing the incoming links.

As in the inter-node distance analysis, churn has only a light effect. Figure 9 shows the number of outgoing links with the same two scenarios as for Figure 7 for an overlay with 10,000 nodes. Again, the 0-percentage on the right-hand side of the graphs is justified by the inter-node distance, which has not been chopped by churn. Moreover, the graphs continue to show the increasing tendency.

We can thus conclude that our join algorithm produces a structure that is quite uniform and regular, which is key to deterministically locate redundant paths and route around failures.

5.2 Routing under Failure-Free Operation

The fact that greedy routing results in big steps at the beginning of the request path limits the number of alternative paths if one node in the request path fails. Conversely, FT-routing proceeds by small steps in the beginning and larger ones in the end. This allows for a larger number of alternative paths until the destination is reached. Intuitively, FT-routing can be seen as striving to "keep all options open" while greedy routing would rather proceed "rushing blindly". This more careful behavior of FT-routing, which is key to ensuring fault-tolerant routing, is analyzed next.

In the following experiments, we consider a system with 10,000 nodes and 20,000 objects in a space of 2^m, $m = 15$, where we issue 200,000 requests. When using greedy routing, a request is sent to the highest node entry smaller than or equal to the requested key.

We have first run experiments in ideal settings without node failures to analyze the path lengths of FT-routing and to compare it against those of greedy routing (GR-routing). We have also included the analysis of LB-routing. Table 1 shows the average and the variance for the path lengths obtained with the three routing strategies in *hypeer*. We note that the results obtained do not differ significantly, which indicates that FT-routing and LB-routing perform well under failure-free operation. The higher average

and variance of FT-routing can be explained by the fact that it can better locate and exploit very short paths, but at the same time some paths are longer than on average because of the incomplete hypercube embedding.

Routing Strategy	Average	Variance
GR-routing	7.27	3.5
LB-routing	7.46	4.5
FT-routing	8.66	24.8

Table 1. Statistics for path length with no failures.

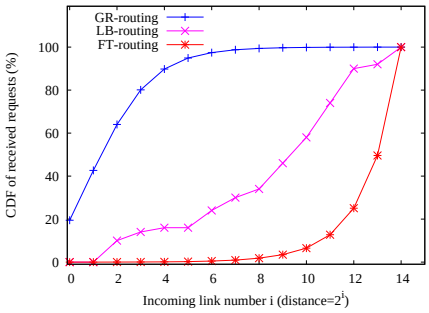

Fig. 10. Comparison between the percentage of received requests per incoming link (2^i) using GR-routing, LB-routing and FT-routing with no failures.

We have then analyzed the average load on the incoming links of the destinations of the issued requests, by repeating the experiment shown in Figure 2 with FT-routing (the results for greedy routing are shown again for comparison purposes). Additionally, we present the results for random routing. Figure 10 shows that, with FT-routing, the incoming links that are used the most are the furthest-away ones from the destination. The reason is that the requests are sent from the source to the closest node that has a direct link to the destination. This trend contrasts with GR-routing, which essentially relies upon close links to destination. LB-routing, where the next hop is chosen at random among the nodes that have a link to the destination, represents a compromise between GR-routing and FT-routing and balances the load on all incoming links.

We have also analyzed the case where the request has to backtrack along predecessors (because of an inappropriate estimation of the inter-node distance). Under the same experiment with 200,000 requests, the real owner of the target is only one or at most two steps away. Moreover, it happens with only a small probability of 6%. This means that the estimation of the inter-node distance performs well.

5.3 Routing upon Failure

To validate the robustness of our routing algorithm, we have run experiments when a given proportion of random nodes fail simultaneously. This adverse scenario simulates correlated failures, e.g., network partitions. We have observed how effective the three routing strategies (GR-routing, LB-routing and FT-routing) are at reaching a given key right after the node failures occur, i.e., before the routing tables have been repaired.

Failure Rates. We deploy two types of experiments under failures. First, we analyze the results for the base algorithms, and next we apply backtracking to each of them. In all experiments we vary the proportion p of failed nodes from 10% to 90%.

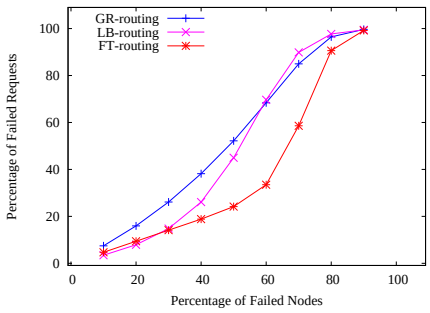

Fig. 11. Failure rates: Comparison between GR-routing, LB-routing and FT-routing, without using backtracking.

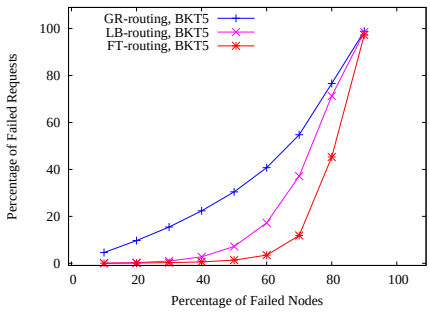

Fig. 12. Failure rates: Comparison between GR-routing, LB-routing and FT-routing, using a backtrack chain of 5 nodes.

Figure 11 shows the comparison between the failure rates of the three routing strategies without using backtracking. For up to 60% node failure, FT-routing has a percentage of failed requests equal to half of the one obtained by GR-routing. From 70% on, the results are still better for FT-routing. As expected, LB-routing has a percentage of failed requests between GR and FT routing. Not surprisingly, at high percentage of node failures, the results are similar for all three routing strategies.

Figure 12 shows the same comparison, but this time all routing strategies use backtracking. In the experiments, we have used a backtrack chain of 5 nodes. The results are obviously better for each of the three routing strategies, again FT-routing obtaining the best results, and LB-routing being in the middle. However, we observe that their results differ much more this time. Backtracking acts much better with FT-routing than with GR-routing, because, when the request is close to the destination and has to backtrack, the jumps back are larger in the case of FT-routing (un-fixing the high order bits), so a larger number of redundant paths can be exploited thereafter. For FT-routing, only a few requests are lost for failure rates of up to p=70%. For instance, with 50% node failure, FT-routing reaches almost always the destination (only 1.4% of the requests fail) while GR-routing can only deliver one third of the requests (30.45%). The low percentage of failed requests for even high rates of failures demonstrates the high dependability of FT-routing.

We have compared our results with the results obtained by Aspnes *et al.* in [17]. The authors showed that backtracking is a good solution to exploit alternative paths, and moreover they applied heuristics to the routing table maintenance. Their results are good, however we obtain better results with FT-routing for up to 70% node failures.

Average Path Length. The average path lengths for the two types of experiments (without and with backtracking) are shown in Figures 13 and respectively 14.

When backtracking is not used (Figure 13), as expected, the path length for GR-routing is the smallest. The increase in the path length of FT-routing is justified by the additional requests (compared to GR-routing) that are successful. For LB-routing, the path length is the longest. This is mostly caused by the random choice of the bits to

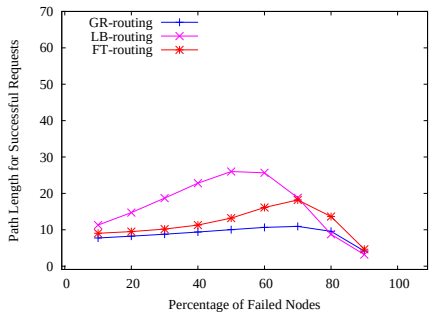

Fig. 13. Average Path length: Comparison between GR-routing, LB-routing and FT-routing under failures, without using backtracking.

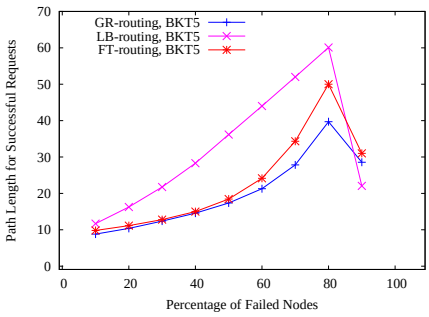

Fig. 14. Average Path length: Comparison between GR-routing, LB-routing and FT-routing under failures, using backtracking with 5 nodes.

be treated, which can lead in some cases to too small order bits to be treated and thus implying hops that are smaller than the inter-node distance.

Backtracking significantly increases the path lengths of all routing strategies (Figure 14), since the jumps back are also counted. The results for FT-routing and GR-routing are getting closer. This can be justified by the requests that backtrack when they are close to the destination. In such cases, GR-routing makes smaller hops than FT-routing, and so it needs more hops to go back to a certain node.

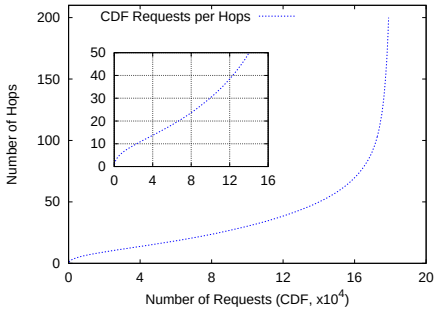

Fig. 15. CDF of the number of requests per number of hops, where 200,000 requests were issued with a maximum acceptable path length of 200 hops and of 50 hops in the inner graph.

Routing	Req Failures	Path avg
FT, max 200 hops	11.86%	34.3
FT, max 50 hops	29.50%	23.0
GR, max 200 hops	54.78%	27.8

Table 2. Percentage of failed requests when using a backtrack chain of 5 nodes and different maximum acceptable path length.

In all experiments, we have set the maximum acceptable path length to 200 hops. Figure 15 shows the cumulative distribution function (CDF) of the number of requests per number of hops under 70% node failures and using a backtrack sequence of 5 nodes (the total number of issued requests is 200,000). In this case, 11.86% of the requests failed (as shown earlier in Figure 12). The average path length is 34.3 hops. There is only a small part of the requests that have very long paths (notice the very quick in-

crease in the path length at the right-hand side of the graph). Thus, we choose to limit the maximum path length to 50 hops (see inner graph), so the requests with path length larger than 50 hops are considered as failed. In this case, the average path length decreases to 23 hops (as expected, since the increase in path length until 50 hops is almost linear), but of course the percentage of failed requests increases. It becomes 29.5%, which is still smaller than in case of GR-routing. Table 2 summarizes these results.

The choice for Redundant Routing. At very high failure rates (80% and 90%) experiments showed that the request failures tend to be independent of the proximity between the source and the destination. However, the request failures experienced by FT-routing for node failures of up to 70% were identified as being mostly due to the proximity between the source and the destination, which severely limits the number of redundant paths. In such cases, our algorithm could be extended to also search for paths that traverse nodes outside the range between source and the destination, i.e., by initially moving away from the destination. For example, for small distances between the source node and the destination, redundant routing could be applied, by sending a request using FT-routing, and another request through the highest routing table entry available, and then apply FT-routing to take advantage of all incoming links of that particular destination. This could also be an alternative solution to using backtracking.

6 Conclusions

The analysis on the fault-tolerant infrastructures and routing strategies shows that the support for fault-tolerance cannot be an afterthought when willing to provide a high fault tolerance at low costs. Greedy routing is simple, however for fault-tolerance, it lacks of dependability.

Consequently, we designed both the infrastructure and the routing strategy of our overlay with the goal to offer the support for fault tolerance. We applied simple modifications to Chord-like systems. Chord cannot easily exploit redundant paths because of its non-determinism in node placement that does not permit treating digits in any order, so we are fixing this by adding some determinism in the placement of the nodes. This means that we may control the position of the nodes on the ring, which is obviously advantageous for the routing strategy. In contrast to the common method of using a hash function to map the nodes on the ring, we approximate a hypercube structure by trying to maintain an even inter-node distance equal to a power of 2 despite churn. Then, we are also modifying the routing protocol to exploit alternative paths by taking into account all possible incoming links of the destination starting from the source node. The rate of request success is much higher, and the maintenance cost remains low, since no additional structures that need to be maintained are required.

Our experiments clearly demonstrate that the FT-routing algorithm, combined with uniform space partitioning that allows us to route deterministically via multiple paths, provide an excellent resilience to failures.

References

1. Kaashoek, M.F., Karger, D.R.: Koorde: A simple degree-optimal distributed hash table. In: Proceedings of the 2nd International Workshop on Peer-to-Peer Systems. (2003) 323–336
2. Malkhi, D., Naor, M., Ratajczak, D.: Viceroy: A scalable and dynamic emulation of the butterfly. In: Proceedings of the 21st ACM Symposium on Principles of Distributed Computing. (2002) 183–192
3. Ratnasamy, S., Francis, P., Handley, M., Karp, R., Shenker, S.: A scalable content addressable network. In: Proceedings of ACM SIGCOMM. (2001) 161–172
4. Stoica, I., Morris, R., Karger, D., Kaashoek, F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: Proceedings of ACM SIGCOMM. (2001) 149–160
5. Rowstron, A., Druschel, P.: Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. Lecture Notes in Computer Science **2218** (2001) 329–350
6. Zhao, B., Huang, L., Stribling, J., Rhea, S., Joseph, A., Kubiatowicz, J.: Tapestry: A resilient global-scale overlay for service deployment. IEEE Journal on Selected Areas in Communications **22**(1) (2004) 41–53
7. Maymounkov, P., Mazieres, D.: Kademlia: A peer-to-peer information system based on the xor metric. In: Proceedings of the 1st International Workshop on Peer-to-Peer Systems. (2002) 53–65
8. Castro, M., Costa, M., Rowstron, A.: Performance and dependability of structured peer-to-peer overlays. In: Proc. 2004 International Conference on Dependable Systems and Networks DSN2004. (2004) 9–19
9. Castro, M., Drushel, P., Ganesh, A., Rowstron, A., Wallach, D.: Secure routing for structured peer-to-peer overlay networks. In: Proc. 5th Symposium on Operating Systems Design and Implementation OSDI2002. (2002) 299–314
10. Locher, T., Schmid, S., Watternhofer, R.: eQuus: A provably robust and locality-aware peer-to-peer system. In: Proceedings of the 6th International Conference on Peer-to-Peer Computing. (2006) 3–11
11. Schlosser, M., Sintek, M., Decker, S., Nejdl, W.: Hypercup – hypercubes, ontologies and efficient search on p2p networks. In: First Workshop on Agents and P2P Computing Springer LNCS 2530. (2002) 112–124
12. Alvarez-Hamelin, J.I., Viana, A.C., Amorim, M.D.: DHT-based functionalities using hypercubes. In: Proceedings of World Computer Congress IFIP WCC. Volume 212. (2006) 157–176
13. Salter, J., Antonopoulos, N.: An efficient fault tolerant approach to resource discovery in p2p networks. Technical Report CS-04-02, University of Surrey Guildford (2004)
14. Wepiwé, G., Simeonov, P.L.: A concentric multi-ring overlay for highly reliable p2p networks. In: NCA. (2005) 83–90
15. Lam, S.S., Liu, H.: Failure recovery for structured p2p networks: Protocol design and performance evaluation. In: Proceedings of ACM SIGMETRICS - Performace. (2004) 199–210
16. Aspnes, J., Diamadi, Z., Shah, G.: Fault-tolerant routing in peer-to-peer systems. In: Proceedings 21st ACM Symp. on Principles of Distributed Computing PODC2002. (2002) 223–232
17. Aspnes, J., Diamadi, Z., Shah, G.: Greedy routing in peer-to-peer systems. extended version of Fault-tolerant routing in peer-to-peer systems (2006)
18. Oh, E., Chen, J.: Parallel routing in hypercube networks with faulty nodes. In: ICPADS. (2001) 338–345