PeerCube: an Hypercube-based P2P Overlay Robust against Collusion and Churn

Emmanuelle Anceaume, Romaric Ludinard, Aina Ravoaja, Francisco Brasiliero

To cite this version:

HAL Id: hal-00916734
https://hal.archives-ouvertes.fr/hal-00916734
Submitted on 13 Dec 2013

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L’archive ouverte pluridisciplinaire HAL, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d’enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.
PeerCube: a Hypercube-based P2P Overlay
Robust against Collusion and Churn*

E. Anceaume, R. Ludinard, A. Ravoaja
IRISA/CNRS/INRIA/ENS Cachan
Campus Universitaire de Beaulieu
Rennes, France
{anceaume,roludina,aravoja}@irisa.fr

F. Brasileiro
Universidade Federal de Campina Grande
Laboratório de Sistemas Distribuídos
58.109-970, Campina Grande, PB, Brazil
fubica@dsc.ufcg.edu.br

Abstract

This paper presents PeerCube, a DHT-based system aiming at minimising performance penalties caused by high churn while preventing malicious peers from subverting the system through collusion. This is achieved by i) applying a clustering strategy to support quorum-based operations; ii) using a randomized insertion algorithm to reduce the probability with which colluding Byzantine peers corrupt clusters, and; iii) leveraging on the properties of PeerCube’s hypercubic structure to allow operations to be successfully handled despite the corruption of some clusters. In spite of a powerful adversary that can inspect the whole system and issue malicious join requests as often as it wishes, PeerCube guarantees robust operations in \(O(\log N)\) messages, with \(N\) the number of peers in the system. Extended simulations validate PeerCube robustness.

1 Introduction

Research on the development of efficient peer-to-peer systems has recently received a lot of attention. This has led to the construction of numerous structured peer-to-peer overlays systems [16,24,19,2,14]. All these systems are based on distributed hash tables (DHTs) which partition an identifier space among all the peers of the system. Structured overlays enjoy numerous important properties. They are efficient, scalable, and tolerant to benign failures. However, less investigation has been carried out for handling both very high churn and collusive behaviour issues. As pointed out by Locher et al. [13], most proposed peer-to-peer overlays are highly satisfactory in terms of efficiency, scalability and fault tolerance when evolving in weakly dynamic environments. On the other hand, in the presence of very frequent connections/disconnections of peers, a very large number of join and leave operations are locally triggered engendering accordingly multiple and concurrent maintenance traffic. Ensuring routing tables consistency quickly becomes unbearable, leading to misrouting, and to possible partitioning of the system. The other fundamental issue faced by any practical open system is the inevitable presence of malicious peers [22]. Guaranteeing the liveness of these systems requires their ability to self-heal or at least to self-protect against this adversity. Malicious peers can devise complex strategies to prevent peers from discovering the correct mapping between peers and data keys. They can mount Sybil attacks [6] (i.e., an attacker generates numerous fake peers to pollute the system), they can do routing table poisoning (also called eclipse attacks [3,22]) by having good peers redirecting outgoing links towards malicious ones, or they can simply drop or re-route messages towards other malicious peers. They can magnify their impact by colluding and coordinating their behaviour.

This paper presents PeerCube, a DHT-based system aiming at avoiding high churn from impacting the performance of the system and at the same time at preventing malicious behaviour (coordinated or not) from subverting the system. As many existing DHT-based overlays, PeerCube is based on a hypercubic topology. PeerCube peers self-organise into clusters whose interconnections form the hypercubic topology. Peers within each cluster are classified into two categories, core members and spares, such that only the former ones are actively involved in PeerCube operations. Thus only a fraction of churn affects the overall topology of the hypercube. Defences against eclipse attacks are based on the observation that malicious peers can more easily draw a successful adversarial strategy from a deterministic algorithm than from a randomised one. We show that regardless of the adversarial strategy colluders employ, the randomised insertion algorithm we propose guarantees that the expected number of colluders in each routing table

\*This work is partially supported by the CAPES / COFECUB program 497-05
is minimal. Furthermore, by keeping the number of core members per cluster small and constant, it allows to rely on the powerful consensus building block to guarantee consistency of the routing tables despite Byzantine peers. Finally, PeerCube takes advantage of independent and optimal length paths offered by the hypercubic topology to decrease exponentially the probability of encountering a faulty peer with the number of independent paths $\mathcal{O}(2^d)$.

To summarise, PeerCube brings together research achievements in both “classical” distributed systems and open large scale systems (Byzantine consensus, clustering, distributed hash tables) so that it efficiently deals with collusion and churn. To the best of our knowledge this work is the first one capable of tolerating collusion by requiring for each lookup, put, join and leave operation $\mathcal{O}(\log N)$ latency and only $\mathcal{O}(\log^2 N)$ messages.

In the remaining of the paper, we discuss related work in Section 2 and then present the system and adversary models in Section 3. Description of the architecture is given in Section 4 together with an analysis of the churn impact. Robustness against malicious behaviours (coordinated or not) is studied in Section 5. Results of simulations are presented in Section 6. We conclude in Section 7.

2 Related Work

In the following, we first review related work that focuses on robustness against malicious peers and then examine policies to handle high churn.

Regarding robustness to malicious behaviour, different approaches have been proposed, each one focusing on a particular adversary strategy. Regarding eclipse attacks, a very common technique, called constrained routing table, relies on the uniqueness and impossibility of forging peers’ identifiers. It consists in selecting as neighbours only the peers whose identifiers are closer to some particular points in the identifier space $\mathbb{Z}$

Some approaches have been successfully implemented into several overlays (e.g., CAN, Chord, Pastry). More generally, to prevent messages from being misrouted or dropped, the seminal works on DHT routing failure tests and redundant routing as a solution to ensure robust routing. Ravoaja and Anceaume extended this approach to cope with colluders by constraining the result of a query, which guarantees to reach the legitimate recipient with high probability $\mathcal{O}(\log^2 N)$ links to other nodes, and require for each operation $\mathcal{O}(\log^3 N)$ messages.

With regard to churn, Li and al. $\mathcal{O}(\log N)$ latency and only $\mathcal{O}(\log^2 N)$ messages.
tocols, we ignore the fact that messages are signed and re-
cipients of a message ignore any message that is not signed
properly. We also use cryptographic techniques to prevent
a malicious peer from observing or unnoticeably modifying
a message sent by a correct peer. However a malicious
peer has complete control over the messages it sends and
receives. Note that messages physically sent between any
two correct peers are neither lost nor duplicated.

4 Architecture Description

As discussed before, our architecture is based on a hy-
percubic topology. The hypercube is a popular intercon-
nection scheme due to its attractive topological properties,
namely, low node degree and low network diameter. Be-
yond these properties, a hypercube offers two important
topological features, namely recursive construction and in-
dependent paths.

4.1 Background

This section presents some preliminaries related to the
hypercubic topology. For more details the reader is invited
to read Saad and Schultz [20]. A $d$-dimensional hypercube,
or $d$-hypercube for short, consists of $2^d$ vertices, where each
vertex $n$ is labelled by its $d$-bits representation. Dimension
$d$ is a fundamental parameter since it characterises both the
diameter and the degree of a $d$-hypercube. Two vertices
$n_0 \ldots n_{d-1}$ and $m = m_0 \ldots m_{d-1}$ are connected by an
edge if they share the same bits but the $i^{th}$ one for some
$i, 0 \leq i < d$, i.e. if their Hamming distance $\mathcal{H}(n, m)$ is
equal to 1. In the following, the notation $n = m'$ stands for
two vertices $n$ and $m$ whose labels differ only by their bit $i$.

Property 1 (Recursive Construction [20]). A $d$-hypercube
can be constructed from lower dimensional hypercubes.

The construction consists in joining each vertex of a
$(d-1)$-hypercube to the vertex of the other $(d-1)$-
hypercubes that is equally labelled, and by suffixing all the
labels of the vertices of the first $(d-1)$-hypercube with 0
and those of the second one with 1. The obtained graph is a
$d$-hypercubr. From this construction, we can derive a sim-
plesurated algorithm for building a $d$-hypercube from a
$(d-1)$ one which involves only 2 messages per link updated
whatever the dimension of the considered system, and thus
has a message complexity of $O(d)$ per peer.

Property 2 (Independent Routes [20]). Let $n$ and $m$ be any
two vertices of a $d$-hypercub.

Two paths are independent if they do not share any com-
mon vertex other than the source and the destination ver-
tices. In a $d$-hypercube, a path from vertex $n$ to vertex $m$ is
obtained by crossing successively the vertices whose labels
are obtained by modifying one by one $n$’s bits to transform
$n$’s label into $m$’s one. Suppose that $\mathcal{H}(n, m) = b$. Then
$b$ independent paths between $n$ and $m$ can be found as fol-
ows: path $i$ is obtained by successively correcting bit $i$, bit
$i+1$, . . . , bit $(i+b-1) \mod b$ among the $b$ different bits
between $n$ and $m$. Note that these $b$ paths are of optimal
length $\mathcal{H}(n, m)$. In addition to these paths, $d-b$ paths of
length $\mathcal{H}(n, m) + 2$ can be constructed as follows: path $j$
of length $\mathcal{H}(n, m) + 2$ is obtained by modifying first bit $j$
on which $n$ and $m$ agree, and then by correcting the $b$ dif-
ferent bits according to one of the $b$ possibilities described
previously, and finally by re-modifying bit $j$.

4.2 PeerCube in a Nutshell

We now present an overview of PeerCube features. Ba-
sically, our architecture has two main characteristics: peers
sharing a common prefix gather together into clusters; and
clusters self-organise into a hypercubic topology.

4.2.1 Clusters

As stated before, each joining peer is assigned a unique ran-
dom ID from an $m$-bit identifier space. Assigning unique
random IDs to peers prevents the adversary from control-
ling a portion of the network, since peers are spread wide
over the network according to their identifier. Peers whose
ID share a common prefix gather together within the same
cluster. Each cluster is uniquely identified with a label that
characterises the position of the cluster in the overall hy-
percubic topology. The label of a cluster is defined as the
shortest common prefix shared by all the peers of that clus-
ter such that the non-inclusion property is satisfied. The
non-inclusion property guarantees that a cluster label nev-
er is never included in the label of another cluster.

Property 3 (Non-Inclusion). If a cluster $C$ labelled with
$b_0 \ldots b_{d-1}$ exists then no cluster $C'$ with $C' \neq C$ whose label
is prefixed with $b_0 \ldots b_{d-1}$ exists.

The length of a cluster label, i.e. the number of bits of
that label, is called the dimension of the cluster. In the fol-
lowing notation $d$-cluster denotes a cluster of dimension
$d$. Dimension determines an upper bound on the number of
links a cluster has with other clusters of the overlay, i.e. the
number of its neighbours. Peers of a $d$-cluster $C$ maintain
a routing table $RT$ such that entry $RT[i]$, with $0 \leq i < d$,
points to peers belonging to one of the $d$ closest clusters.
to \( C \). (Distance notion is detailed in Section 4.2.2) References to clusters that point toward \( C \) are maintained by \( C \)’s members in a predecessor table \( PT \). Note that maintaining such a data structure is not mandatory, i.e., those clusters can be easily found by the topological properties of PeerCube. However, keeping this information makes the maintenance operations more efficient. Regarding data, all the peers of a cluster are responsible for the same data keys and their associated data. As for most existing overlays, a data key is placed on the closest cluster to this key. Placing a data key on all the peers of a cluster naturally improves fault tolerance since this increases the probability that this key remains available even if some of the peers fail. To keep this probability high, the size of a cluster must not undershoot a certain predefined value \( S_{\min} \), which depends on the probability of peers’ failures. Finally, for scalability reasons, each cluster size is upper bounded by a constant value \( S_{\max} \) specified later on.

### 4.2.2 Hypercubic Topology

Clusters self-organise into a hypercubic topology, such that the position of a cluster into the hypercube is determined by its label. Ideally the dimension of each cluster \( C \) should be equal to some value \( d \) to conform to a perfect \( d \)-hypercube. However, due to churn and random identifier assignment, dimensions may differ from one cluster to another. Indeed, as peers may join and leave the system asynchronously, cluster \( C \) may grow or shrink more rapidly than others. In the meantime, bounds on the size of clusters require that, whenever the size of \( C \) exceeds \( S_{\max} \), \( C \) splits into clusters of higher dimensions, and that, whenever the size of \( C \) falls under \( S_{\min} \), \( C \) merges with other clusters into a single new cluster of lower dimension. Finally, since peers IDs, and thus cluster labels, are randomly assigned, setting some of the labels may initially not be represented at all. For all these reasons dimensions of clusters may not be homogeneous.

To keep the structure as close as possible to a perfect hypercube and thus to benefit from its topological properties, we need a distance function \( D \) that allows to uniquely characterise the closest cluster of a given label. This is obtained by computing the numerical value of the “exclusive or” (XOR) of cluster labels \( \overline{b} \). To prevent two labels to be at the same distance from a given bit string, labels are suffixed with as many bits “0” as needed to equalise their size to \( m \).

**Definition 1** (Distance \( D \)). Let \( C = a_0 \ldots a_{d-1} \) and \( C' = b_0 \ldots b_{d-1} \) be any two \( d \) (resp. \( d' \)) -clusters:

\[
D(C, C') = D(a_0 \ldots a_{d-1}0^{m-d}, b_0 \ldots b_{d-1}0^{m-d}) = \sum_{i=0}^{m-1} a_i \overline{b}_i 2^{m-i}
\]

Distance \( D \) is such that for any point \( p \) and distance \( \Delta \) there is exactly one point \( q \) such that \( D(p, q) = \Delta \) (which does not hold for the Hamming distance). Finally, labels that have longer prefix in common are closer to each other. We are now ready to detail the content of a cluster’s routing table. Let \( C = b_0 \ldots b_{d-1} \) and \( C^\prime = b_0 \ldots b'_1 \ldots b_{d-1} \). Then, \( C \)'s \( i \)th neighbour in PeerCube is cluster \( C' \) whose label is the closest to \( C^\prime \).

**Property 4.** Let \( C \) be a \( d \)-cluster. Then, \( \forall i, 0 \leq i < d \), entry \( i \) of the routing table of \( C \) is cluster \( C' \) such that for each cluster \( C'' \neq C' \), \( D(C^\prime, C') < D(C^\prime, C'') \) holds.

By the distance \( D \) definition, it is easy to see that if for each cluster \( C \) in PeerCube the distance between \( C^\prime \) and its \( i \)th neighbour is equal to 0 (with \( 0 \leq i < d \)), then PeerCube maps a perfect \( d \)-hypercube. From Property 4 we have:

**Lemma 1.** Let \( C = b_0 \ldots b_{d-1} \) be a \( d \)-cluster. Then \( \forall i, 0 \leq i < d \), \( C \)'s \( i \)th neighbour is cluster \( C' \) such that \( C' \) is prefixed with \( b_0 \ldots b_i \) if such a cluster exists. Otherwise, \( C' = C \).

This can be seen by observing that, by definition of \( D \), \( C' \) shares the longest prefix with \( C^\prime \), that is at least the prefix \( b_0 \ldots b_1 \). Otherwise \( C' \) would be the closest cluster to \( C^\prime \). We exploit this property to construct a simple lookup protocol which basically consists in correcting the bits of the source towards the destination from the left to the right.

### 4.3 Leveraging the Power of Clustering

**Dimensions Disparity** As described before, clusters dimensions are not necessarily equal to each other. By simply setting \( S_{\max} > \log_2 N \), we can make the dimensions disparity small and constant. Indeed, observe that the dimension of a cluster is necessarily greater than or equal to \( \log_2 \frac{N}{S_{\max}} \). This follows from the fact that the minimum number of clusters is \( N/S_{\max} \), which determines the minimum number of bits needed to code the label of a cluster. Furthermore, by setting \( S_{\max} > \log_2 N \), we can show by using Chernoff’s bounds that the dimension of a cluster is w.h.p. lower than \( \log_2 \frac{N}{S_{\max}} + 3 \). Indeed, since labels are uniformly randomly assigned, setting \( S_{\max} \) to a higher value decreases clusters dimension. Thus distance \( \delta \) between any two clusters dimensions is w.h.p. less than or equal to 3. Furthermore the number of non-represented prefixes is at most \( 2^\delta \), which is very small with regard to the total number of clusters \( N/S_{\max} \). Consequently, by setting \( S_{\max} > \log_2 N \), PeerCube is very close to a \( (\log_2 \frac{N}{S_{\max}}) \)-hypercube, which guarantees PeerCube to enjoy the attractive topological properties of a perfect hypercube of diameter \( \log_2 \frac{N}{S_{\max}} \). Henceforth \( S_{\max} \) is in \( \Theta(\log N) \).

---

1. In the following, with high probability (w.h.p.) means with probability greater than \( 1 - \frac{1}{N} \).
2. Note that for a pure hypercube, the dimension disparity is \( \log_2 N \).
Limiting the Impact of Churn  We have just shown that by having peers self-organised in a hypercube of clusters we get w.h.p. an overlay of diameter $O(N \log N)$. We now describe how peers take advantage of that clustering to limit the impact of churn on the overall system. Specifically, peers within a cluster are classified into two categories: core and spare members. Only core members are in charge of PeerCube operations (i.e., inter clusters message forwarding, routing table maintenance, computation of cluster view membership, and keys caching). Size of the core set is equal to the minimal size of a cluster, i.e., constant $S_{\text{min}}$. Core members form a clique, i.e., they point to each other. View of the core set is denoted $V_c$. In contrast to core members, spare members are temporarily inactive, in the sense that they are not involved in any of the overlay operations. They only maintain links to a subset of core members of their cluster and cache the set of keys and associated data as core members do. Within a cluster, apart from the core members that maintain the view $V_c$ of the spares set, no other peer in the system is aware of the presence of a particular spare, not even the other spares of the cluster. As a consequence, routing tables only point to core members, that is $S_{\text{min}}$ references per entry are needed.

Achieving High Consistency  By keeping the size of the core set to a small and constant value, we can afford to rely on the powerful consensus building block to guarantee consistent routing tables among correct core members despite the presence of a fraction $\mu$ of Byzantine peers among them. Briefly, in the consensus problem, each process proposes a value, and all the non-faulty processes have to eventually decide (termination property) on the same output value (agreement property), this value having been proposed by at least one process (validity property). Various Byzantine consensus algorithms have been proposed in the literature (good surveys can be found in [8, 9]). In PeerCube, we use the solution proposed by Kotla et al. [11] essentially because it provides optimal resiliency, i.e. tolerates up to $\frac{n-1}{3}$ Byzantine processes in a group of $n$ processes, and guarantees that a value proposed only by Byzantine processes is never decided by correct ones. Moreover, message complexity is in $O(n^2)$ in the worst case, and $O(n)$ in executions where Byzantine processes are not present. Note that in our context, $n = S_{\text{min}}$.

4.4 PeerCube Operations

From the application point of view, three key operations are provided by the system: the lookup($k$) operation which enables to search for key $k$, the join operation that enables a peer to join the system, and the leave operation, indicating that some peer left the system. Note that the put($x$) operation, that enables to insert data $x$ in the system, is not described since it is very similar to the lookup() operation. From the topology structure point of view, three events may result in a topology modification: when the size of a cluster exceeds $S_{\text{max}}$, this cluster splits into two new clusters; when the size of a cluster goes below $S_{\text{min}}$, this cluster merges with other clusters to guarantee the cluster resiliency; finally, when a peer cannot join any existing cluster because none of them matches the peer identifier prefix, then a new cluster is created. For robustness reasons, a cluster may have to temporarily exceed its maximal size $S_{\text{max}}$ before being able to split into two new clusters. This guarantees that resiliency of both new clusters is met, i.e. both clusters sizes are at least equal to $S_{\text{min}}$. A similar argument applies to the create operation. For this specific operation, peers whose identifiers do not match any cluster label, temporarily join the closest cluster to their identifier, and whenever $S_{\text{split}} \geq S_{\text{min}}$ temporary peers share the same prefix then they create their new cluster. Threshold $S_{\text{split}}$ is discussed in Section 4.4.2. These three additional operations exploit the recursive construction property of hypercubes to minimise topology changes, and rely on the Byzantine-consensus building block to achieve high consistency among routing tables. For space reasons, description of these operations are not presented in the paper. However, each of them is detailed in the companion paper [11].

4.4.1 lookup Operation

In this section we describe how peer $p \in \mathcal{C}$ locates a given key $k$ through the lookup operation. Basically, locating $k$ consists in walking in the overlay by correcting one by one and from left to right the bits of $p$’s identifier to match $k$. By Lemma 1 and by distance $D$, this simply consists in recursively contacting the closest cluster to $k$. In a failure free environment, this operation would be similar to a typical lookup operation, except that if the originator $p$ of the lookup was a spare member, then $p$ would forward its request to a randomly chosen core member of $\mathcal{C}$. Then the request would be propagated until finding either a peer of a cluster labeled with a prefix of $k$, or no cluster closer to $k$ than the current one. The last contacted peer would return to the originating peer $p$ either the requested data if it exists, or null otherwise. Now, suppose that malicious peers may drop or misroute requests they receive to prevent them from reaching their legitimate destination. We adapt the lookup operation by using the width path approach, commonly used in fault tolerant algorithms, which consists in forwarding a request to sufficiently enough peers so that at least one correct peer receives it. This is described in Figure 1. Specifically, a request is forwarded to $(S_{\text{min}} - 1)/3 + 1$ randomly chosen core members of the closest cluster to the request destination, instead of only one randomly chosen core member as in the basic lookup op-
Upon $\text{lookup}(k)$ from the application do
  
  if ($p$.type $\neq$ {core}) then
      
      $\{q_0 \ldots q_{\lfloor(S_{\text{min}}-1)/3\rfloor}\} \leftarrow p$.coreRandomPeer();
      
      $p$ sends (LOOKUP, $k$, $p$) to $\{q_0 \ldots q_{\lfloor(S_{\text{min}}-1)/3\rfloor}\}$
  
  else
      
      $C \leftarrow p$.findClosestCluster($k$);
      
      $p$ sends (LOOKUP, $k$, $p$) to a random subset of
      
      $\lfloor(S_{\text{min}}-1)/3\rfloor + 1$ peers in $C$.coreSet;
  
enddo

Upon receiving (LOOKUP, $k$, $q$) from the network do
  
  $C \leftarrow p$.findClosestCluster($k$);
  
  if ($p$.cluster.label $=$ $C$) then
      
      $p$ sends (LOOKUP, $k$, $q$) to core members in $C$
      
      if not already done;
  
      data $\leftarrow k$’s data if cached otherwise null;
      
      sends ($k$.data) to the originating $q$ by using the reverse path;
  
  else
      
      $p$ sends (LOOKUP, $k$, $q$) to a random subset of
      
      $\lfloor(S_{\text{min}}-1)/3\rfloor + 1$ peers in $C$.coreSet;
  
enddo

findClosestCluster($k$)

if ($p$.dim$=0$ or $p$.cluster.prefix($k$)) then
    
    $C \leftarrow p$.cluster;
  
else
    
    $C$’s label $\leftarrow RT_p(0)$.label;
    
    for ($i = 0$ to $p$.dim $-$ 1) do
      
      if $(D(k,RT_p(i)$.label) $<$ $(D(k,C$.label))) then
        
        $C$’s label $\leftarrow RT_p(i)$.label;
      
  return $C$;

Figure 1. $\text{lookup}$ Operation at Peer $p$

Upon $\text{join}(p)$ from the application do

$\{q_0 \ldots q_{\lfloor(S_{\text{min}}-1)/3\rfloor}\} \leftarrow \text{findBootstrap}();

$p$ sends (JOIN, $p$) to $q$ in $\{q_0 \ldots q_{\lfloor(S_{\text{min}}-1)/3\rfloor}\}$;

enddo;

Upon receiving (JOIN, $q$, $p$) from the network do

$C \leftarrow p$.findClosestCluster($q$.id);

if ($p$.cluster $=$ $C$) then

if ($p$.cluster.prefix($q$.id)) then

$p$ broadcasts (JOIN, $q$.id) to $p$’s core set;

else

$p$ broadcasts (JOIN, $q$.id) to $p$’s core set;

else

$p$ sends (JOIN, $q$.id) to a random subset of

$\lfloor(S_{\text{min}}-1)/3\rfloor + 1$ peers in $C$’s core set;

enddo;

Upon delivering (JOIN, $q$.id) from the network do.

$V_s \leftarrow V_s \cup q$;

if ($p$.clusterIsSplit) then $p$.split();

$\mathcal{N} = p$.findClosestCluster($q$.id);

$p$ sends (JOIN, $\mathcal{N}$.state) to $q$;

enddo;

Upon delivering (JOIN, $q$.id) from the network do;

$p$.temp $\leftarrow p$.temp $\cup q$;

if ($p$.tempIsSplit) then $p$.create($p$.temp);

$\mathcal{C} = p$.findClosestCluster($q$.id);

$p$ sends (JOIN, $\mathcal{N}$.state) to $q$;

enddo;

Figure 2. $\text{join}$ Operation at Peer $p$

Operation. In addition, in the last contacted cluster $C$, when a core member $p \in C$ receives the request, if $p$ has not already sent it to all core members of $C$ then it does so and returns the response through the reverse path. Hence, each peer that forwarded the request waits for a quorum of responses (i.e., $\lfloor(S_{\text{min}}-1)/3\rfloor + 1$) before propagating the response back in the reverse path. When the originator $q$ of the $\text{lookup}$ request receives $\lfloor(S_{\text{min}}-1)/3\rfloor + 1$ similar responses ($k$.data, $C$) issued from peers whose ID prefix matches the one $q$ initially contacted, then $q$ can safely use the received data. Otherwise, $q$ discards it. It is easy to see that if there are no more than $\lfloor(S_{\text{min}}-1)/3\rfloor$ malicious core members per cluster crossed, then a $\text{lookup}$ operation invoked by a correct peer returns the legitimate response.

Lemma 2. The $\text{lookup}(k)$ operation returns the data associated to $k$ if it exists, null otherwise. This is achieved in $\mathcal{O}(\log N)$ hops and requires $\mathcal{O}(\log N)$ messages.

For space limitations, proofs of lemmata are omitted from the paper. However, they are available in [1].

4.4.2 $\text{join}$ Operation

Recall that by construction each cluster $C$ contains all the core and spare members $p$ such that $C$’s label is a prefix of $p$’s ID, and that each peer $p$ belongs to a unique cluster. To join the system, peer $p$ sends a $\text{join}$ request to a correct peer it knows in the system. The request is forwarded until finding the closest cluster $C$ to $p$’s ID. Two cases are possible: either $C$’s label matches the prefix of $p$’s ID or the cluster $\mathcal{N}$ of $p$ should be inserted into does not already exist ($C$ is only the closest cluster to $\mathcal{N}$). In the former case, $p$ is inserted into $C$ as a spare member. Inserting newcomers as spare members prevent malicious peers from designing deterministic strategies to increase their probability to act as core member. In the latter case, $p$ is temporarily inserted into $\mathcal{N}$ until creation of $\mathcal{N}$ is possible, i.e., predicate $\text{tempIsSplit}()$ in Figure 2 holds. This predicate holds if there exist $S_{\text{split}}$ temporary peers in $C$ that share a common prefix. Note that temporary peers do not participate in the cluster life (they do not even cache data, contrary to spares), and only core members are aware of their presence. Threshold $S_{\text{split}}$ is introduced to prevent the adversary from triggering a “split-merge” cyclic phenomenon. Indeed, a strong adversary can inspect the system and locate the clusters that are small enough so that the departure of malicious peers from that cluster triggers a merge operation with other clusters, and their re-joining activates a split operation of the newly created cluster. Thus by setting $S_{\text{split}} = S_{\text{min}} > \lceil\frac{\text{max}_{\text{mag}}-1}{3}\rceil$ with $\text{max}_{\text{mag}} = \lceil\frac{\text{max}_{\text{mag}}-1}{3}\rceil$ the expected number of malicious peers in a cluster, probability of this phenomenon is negligible. In both cases, i.e. whether $p$ is inserted as spare or temporary peer of $C$, $p$’s insertion
is broadcast to all core members. The broadcast primitive guarantees that if a correct sender broadcasts some message \( m \), then all correct recipients eventually deliver \( m \) once\(^4\).

Upon \( q \)’s insertion by staying in place; this way, core sets within the spare set) does not prevent the adversary from polluting the core set, the whole component guarantees that if a correct sender broadcasts some message \( m \), then all correct recipients eventually deliver \( m \) once\(^4\).

Thus, when core members detect that \( q \)’s departure has to be detected by \( C \). The join operation is insensitive to collusion. That is if before a join operation in \( C \) the expected number of malicious peers in \( C \) is \( \mu S_{\text{min}} \), then after a join in \( C \) the expected number of malicious peers is still equal \( \mu S_{\text{min}} \).

**Lemma 3.** The join operation is insensitive to collusion. That is if before a join operation in \( C \) the expected number of malicious peers in \( C \) is \( \mu S_{\text{min}} \), then after a join in \( C \) the expected number of malicious peers is still equal \( \mu S_{\text{min}} \).

**4.4.3 Leave Operation**

The leave operation is executed when a peer \( q \) wishes to leave a cluster or when \( q \)’s failure has been detected. Note that in both cases, \( q \)’s departure has to be detected by \( \lceil (2S_{\text{min}} + 1)/3 \rceil + 1 \) core members so that a malicious peer cannot abusively pretend that some peer \( q \) left the system. Thus, when core members detect that \( q \) left, two scenarios are possible. Either \( q \) belonged to the spare set, in which case, core members simply update their spare view to reflect \( q \)’s departure, or \( q \) belonged to the core set. In the latter case, \( q \)’s departure has to be immediately followed by the core view maintenance to ensure its resiliency (and thus the cluster resiliency). To prevent the adversary from devising collusive scenario to pollute the core set, the whole composition of the core set has to be refreshed. Indeed, replacing the peer that left by a single one (even randomly chosen within the spare set) does not prevent the adversary from ineluctably corrupting the core set: once malicious peers succeed in joining the core set, they maximise the benefit of their insertion by staying in place; this way, core sets are eventually populated by more than \( \lceil 2S_{\text{min}}/3 \rceil \) malicious peers, and thus become – and remain – corrupted. This is illustrated in Section 5. Thus each core member chooses \( S_{\text{min}} \), random peers among both core and spare members, and proposes this subset to the consensus. By the consensus properties, a single decision is delivered to all core members, and this decision has been proposed by at least one correct core member. Thus core members agree on a unique subset which becomes the new core set. Note that in addition to preventing collusion, refreshing the whole core set guarantees that the expected number of malicious peers in core sets, and thus the number of corrupted entries in routing tables is bounded by \( \mu S_{\text{min}} \) which is minimal:

**Lemma 4.** After a core member’s departure, the expected number of malicious peers in that core is at most \( \mu S_{\text{min}} \).

**Lemma 5.** Upon a core member’s departure, for any randomized algorithm, there exists an adversarial strategy such that the expected number of malicious peers in the core is at least \( \mu S_{\text{min}} \).

Remark that because of the asynchrony of the system, some of the agreed peers \( s \) may still belong to some views while having been detected as failed or left by others, or may belong to only some views because of their recent join. In the former case, all the correct core members eventually deliver the consensus decision notifying \( s \)’s departure, and new consensus is run to replace it. Note that for efficiency reason, each core member can ping the peers it proposes before invoking the consensus. In the latter case, \( s \)’s recent arrival is eventually notified at all correct core members by properties of the broadcast primitive (see join operation), and thus they insert \( s \) in \( V_S \). Then each core member \( p \) notifies all the clusters that point to \( C \) (i.e. entries of \( p \)’s PT table) of \( C \)’s new core set. Core members of each such cluster can safely update their entries upon receipt of \( \lceil 2S_{\text{min}}/3 \rceil + 1 \) similar notifications. This is encapsulated into the leavePredTable() procedure in Figure 3. Similarly, all the peers \( \{s_1, . . . , s_{\text{min}}\} \) are safely notified about their new state, and locally handle the received data structures (invocation of leaveRoutingTable() procedure). Former core members only keep their keys and the associated data. In all cases a constant number of messages are exchanged for a leave.

**5 Handling Collusion**

**5.1 Thwarting Eclipse Attacks**

An eclipse attack enables the adversary to control part of the overlay traffic by coordinating its attack to infiltrate routing tables of correct peers. As shown in the previous section, PeerCube operations thwart those attacks es-

---

\(^4\)PeerCube relies on the asynchronous Byzantine-resistant reliable broadcast of Bracha \(^2\), whose time complexity is in \( O(1) \) and message complexity is in \( O(n^2) \). As for consensus, in our case \( n = S_{\text{min}} = \text{cst.} \)

---

**Figure 3.** leave Operation at Peer \( p \)
sentially by preventing colluders from devising deterministic strategies to join core sets (i.e., newcomers are inserted as spare members) and by reaching agreement among core members on any event that affects PeerCube topology. Correctness of these operations relies on the hypothesis that no more than \( \lceil \frac{S_{\text{max}}}{2} \rceil \) malicious peers populate core sets, that is the fraction of malicious peers in any core set is no more than 1/4. Probability that such an assumption does not hold is now discussed. Let us first compute the upper bound on the probability to corrupt a core set. This holds when the clusters number is minimal (i.e. equal to \( \frac{S_{\text{max}}}{2} \)). Denote by \( X_u \) the random variable describing the number of malicious peers in a cluster, and by \( Y_u \) the random variable describing the number of malicious peers in a core. Clearly, \( Y_u \) depends on \( X_u \). Since identifiers are randomly chosen, inserting malicious peers into clusters can be interpreted as throwing \( \mu N \) balls one by one and randomly into \( \frac{S_{\text{max}}}{N} \) bins. The probability that \( x \) balls (malicious peers) are inserted into a bin (cluster) is 
\[
P(X_u = x) = \left( \frac{\mu N}{S_{\text{max}}} \right)^x (1 - \frac{\mu N}{S_{\text{max}}})^{N-x}.
\]
By the leave-leave operation, each departure from a core set is followed by the rebuilding of this set with \( S_{\min} \) randomly chosen peers among the \( S_{\text{max}} \) peers of the cluster. This can be interpreted as picking simultaneously \( S_{\min} \) balls among \( S_{\text{max}} \) balls among which \( x \) are black (malicious peers) and \( S_{\text{max}} - x \) are white (correct ones). Thus, the probability of having \( y \) malicious peers inserted in the core, knowing the number of malicious peers \( x \) in the cluster, is given by 
\[
P(Y_u = y | X_u = x) = \left( \frac{\mu N}{S_{\text{max}}} \right)^y (1 - \frac{\mu N}{S_{\text{max}}})^{S_{\text{max}}-y}.
\]
Finally, the tight upper bound on the corruption probability is equal to 
\[
p_u = 1 - \sum_{y=0}^{\left\lfloor \frac{S_{\text{max}}}{2} \right\rfloor} P(Y_u = y | X_u = x) P(X_u = x).
\]
Proceeding as above, the tight lower bound on the corruption probability is 
\[
p_l = 1 - \sum_{x=0}^{\left\lfloor \frac{S_{\text{max}}}{2} \right\rfloor} P(Y_u = y | X_u = x) P(X_u = x).
\]
We can now derive upper and lower bounds on the probability that a request reaches its legitimate destination. The probability that the number of hops of a request be \( h \) is equal to 
\[
d_{\text{max}} \left( \frac{1}{2} \right)^{d_{\text{max}}} (1 - p_u)^h, \text{ with } d_{\text{max}} = \log_2 \left( \frac{N}{S_{\text{max}}} \right) + 3 \text{ the maximal dimension of a cluster.}
\]
Such a request is successful if none of the \( h \) clusters crossed by this request are corrupted. Thus its probability of success is at least 
\[
\sum_{h=0}^{d_{\min}} \left( \frac{1}{2} \right)^{d_{\min}} (1 - p_u)^h, \text{ and at most } \sum_{h=0}^{d_{\min}} \left( \frac{1}{2} \right)^{d_{\min}} (1 - p_l)^h, \text{ with } d_{\min} = \log_2 \left( \frac{N}{S_{\text{max}}} \right).
\]
Recall that the policy we propose to replace a left core member is to refresh the whole composition of the core set by randomly choosing peers within the cluster. We opposed this policy to the one which consists in replacing the core member that left by a single one randomly chosen in the cluster (see Section 4.4.3). Figure 4 compares the lower bound on the probability of successful requests with these two policies according to \( S_{\text{max}} \), for different ratio of malicious peers in the system, and considering \( N = 1,000 \). The first observation is that probability of success for the policy we propose (labelled by w/ with randomisation in the Figure) varies lightly with \( S_{\text{max}} \) value. This confirms that setting \( S_{\text{max}} > O (\log N) \) does not bring any additional robustness to PeerCube. The second observation is that for the second policy (denoted by w/o randomisation), that probability drastically decreases with increasing values of \( S_{\text{max}} \), even for small values of \( \mu \). This corroborates the weakness of such a policy in presence of a strong adversary.

![Figure 4. Probability of success of requests w.r.t. \( S_{\text{max}} \)](image)

5.2 Robust Routing through Independent Routes

We have just seen that because identifiers are randomly assigned, the ratio of malicious peers in some clusters may exceed the assumed ratio \( \mu \) of malicious peers in the system, and thus may impact the resilience of PeerCube. Since pollution increases with \( S_{\min} \), a possible solution to increase the resilience is to augment \( S_{\min} \) according to \( S_{\text{max}} \), i.e. to have \( S_{\min} \) in \( O (\log N) \). However, because of the Byzantine resistant consensus this makes maintenance operations cost in \( O (\log^3 N) \) or in \( O (\log^2 N) \) because of the broadcast primitive. To circumvent this issue, we extend Castro et al. 3 approach by sending a request over independent routes. We adapt the independent routes construction algorithm presented in Section 4.4 to match PeerCube features. Essentially, the search is adapted to find the closest cluster to the theoretical one when this latter one does not exist. Denote by \( b \) the number of bit differences between \( p \)'s identifier, the source of the request, and \( q \)'s identifier, the destination peer. Recall that the \( i \)th route is obtained by successively correcting bits \( p_i, p_{i+1}, \ldots, p_{(i+b-1) \mod b} \) for \( 0 \leq i \leq b-1 \), with \( p_i, p_{i+1}, \ldots, p_{(i+b-1) \mod b} \) the position of the \( b \) bits that differ between \( p \) and \( q \). We modify this pro-
Lemma 6. The independent routes algorithm finds at least 
$\log_2 \frac{N}{S_{\text{max}}}$ independent routes of length $O(\log N)$ with probability $p$. 

We now examine the probability $p_{\text{succ}}$ for a request issued by a correct peer to reach its legitimate destination when that request is sent over $r$ independent routes of length $h$, with $d_{\text{min}} \leq r \leq d_{\text{max}}$. The request is successful if at least one route does not contain any corrupted cluster. Let $p$ denote the exact probability that a cluster is corrupted, i.e. $p_t \leq p \leq p_u$. The probability of success of a request using $r$ independent routes of length $h$ is $1 - (1 - (1 - p)^h)^r$. Thus probability $p_{\text{succ}}$ is lower bounded by 
\[
\sum_{h=0}^{d_{\text{max}}+2} \left( \frac{d_{\text{max}}+2}{h} \right) \left( \frac{1}{2} \right)^{d_{\text{max}}+2} (1 - (1 - p)^h)^r,
\]
and upper bounded by 
\[
\sum_{h=0}^{d_{\text{min}}} \left( \frac{d_{\text{min}}}{h} \right) \left( \frac{1}{2} \right)^{d_{\text{min}}} (1 - (1 - p)^h)^r.
\]
Term $d_{\text{max}} + 2$ in the first equation comes from the non-optimal paths of the independent routing algorithm. Figure 5 shows the remarkable increase in PeerCube robustness when using independent routes w.r.t. to a single route. Note also that whatever the percentage of malicious peers in the system the probability of success degrades gracefully (logarithmically) with respect to $N$.

6 Simulation

In this section, we present the results of an experimental evaluation of PeerCube performed on PeerSim a simulation platform for P2P protocols. The simulation is event based. The workload is characterised by the number of and arrival/departure pattern of peers and by the distribution of requests they issue. Each experiment uses a different workload.

Churn Impact In these experiments, we study the ability of PeerCube to greatly reduce the impact of high dynamics on peers load. In particular, we analyse the benefit drawn from appointing newcomers as spare members on the number of routing tables updates. In Figure 6, the number of routing tables updates in a network of up to 10,000 peers is depicted. Bursts of joins and leave are cyclically generated (every 500 simulation time unit, up to 500 peers issue join or leave operations). $S_{\text{max}} = 13$, and $S_{\text{min}} = 4$. A failure-free environment is assumed. The dotted curve shows the number of triggered routing tables updates in a cluster-based hypercubic topology in which all clusters members actively participate in the overlay operations (denoted by PeerCube without core/spare classification in the figure), while the solid curve depicts the number of routing tables updates generated in PeerCube (denoted by PeerCube). As expected, using newcomers as hot spares drastically reduces the number of routing tables updates for both joins and departures events. For instance, the burst of joins generated during simulation time 27,000 and 27500 have triggered no routing tables updates for PeerCube while it has given rise to 50,400 updates for PeerCube without core/spare classification.

Robustness against Collusion In these experiments, we test the ability of PeerCube to achieve a robust lookup operation despite the presence of a strong adversary. As described in the previous section, robust lookup is realized by two techniques. First, by preventing malicious peers from strategizing to get inserted within core sets; through the randomization insertion algorithm, we minimize the ratio of
malicious peers into routing tables. Second, by taking advantage of independent and optimal length paths offered by the hypercubic topology to guarantee that a request sent by a correct peer reaches its legitimate destination with probability close to 1. Figure 7 shows for $N = 1,000$ peers, the probability of successful requests sent by correct peers w.r.t. to the ratio of malicious peers in the system. The main observation is that experiments fully validate theoretical results. Namely, for up to 15% of malicious peers, 98% of the requests issued from correct peers are successful, and for 25% of malicious peers, in average, 90% of the requests are successful, which clearly emphasises PeerCube robustness to co-ordinated malicious behaviour.

7 Conclusion

In this paper we have presented PeerCube, a DHT-based system that is able to handle high churn and collusive behavior. Many existing P2P systems exhibit some fault tolerance or churn resiliency. The main contribution of PeerCube is to combine existing techniques from classical distributed computing and open large distributed systems in a new way to efficiently decrease churn impact and to tolerate collusion of malicious peers as shown analytically and validated through experimental simulation. For future work, we are planning to study strategies against a computationally unbounded adversary, that is an adversary, beyond being able to inspect the whole system and issue join and leave requests as often it wishes (as studied in this paper), can carefully choose the IDs of the Byzantine peers, so that it can place them at critical locations in the network [7][21].

Acknowledgment

We would like to thank the CAPES-COFECUB program 497-05 for having partially supported this work.

References


