Approximate Matching for Peer-to-Peer Overlays with Cubit

Bernard Wong[*]
bwong@cs.cornell.edu

Aleksandrs Slivkins[†]
slivkins@microsoft.com

Emin Gün Sirer[*]
egs@cs.cornell.edu

Abstract

Keyword search is a critical component in most content retrieval systems. Despite the emergence of completely decentralized and efficient peer-to-peer techniques for content distribution, there have not been similarly efficient, accurate, and decentralized mechanisms for content discovery based on approximate search keys. In this paper, we present a scalable and efficient peer-to-peer system called Cubit with a new search primitive that can efficiently find the k data items with keys most similar to a given search key. The system works by creating a keyword metric space that encompasses both the nodes and the objects in the system, where the distance between two points is a measure of the similarity between the strings that the points represent. It provides a loosely-structured overlay that can efficiently navigate this space. We evaluate Cubit through both a real deployment as a search plugin for a popular BitTorrent client and a large-scale simulation and show that it provides an efficient, accurate and robust method to handle imprecise string search in filesharing applications.

1 INTRODUCTION

Peer-to-peer data distribution techniques have recently become widely deployed because they are efficient, scalable and resilient to attacks. Recent studies indicate that at least 71% of the data volume on long-haul links is due to peer-to-peer filesharing applications [35]. Yet locating content in a peer-to-peer system poses significant problems. Imprecision stemming from partial specifications of keywords, common variations of search terms and misspellings are common. For instance, approximately 20% of all Google queries for "Britney Spears" misspell the artist's name [1]. Efficiently routing a query to a set of objects whose keys are close but not identical to the search key is a difficult problem known as *approximate matching*.

Modern peer-to-peer substrates do not provide efficient primitives for approximate matching. Unstructured peer-to-peer systems such as [3] provide a *search* primitive, which is typically based on query broadcast [1]. Gnutella nodes receiving the search query match it against their database of known items using a fuzzy similarity metric to yield approximate matches. Such broadcast-based approaches are inefficient as they may take up to N hops in the worst case, where N is the number of hosts, and place a superlinear aggregate load on the network. In contrast, structured peer-to-peer systems [41, 43, 50, 37, 32, 27] provide an efficient *lookup* primitive that can typically locate a target within $O(\log N)$ hops. While these systems provide strong worst-case bounds, the lookup operation does not permit approximate matching. Naive approaches to layer approximate matching on top of a DHT lookup, by inserting each object under all possible key variations or performing every query in parallel with all variants of the search key, lead to highly inefficient solutions. Systems that permit *range lookups* [13, 18] can perform a lookup within a range defined by numeric coordinates, but are difficult to adopt for use with approximate string matching. Overall, existing systems provide inefficient and approximate search or efficient and precise lookup, but not efficient and approximate match. As a result, the highly popular BitTorrent distribution mechanism still relies on centralized components called torrent aggregators for the initial search, rendering it vulnerable to a variety of attacks. For example, PirateBay, the world's largest torrent aggregator, has been the target of attacks by hackers [7], planned attacks by an anti-piracy corporation [6], and was forced to shut down temporarily by law enforcement agencies [5].

In this paper, we present Cubit, a scalable peer-to-peer system that can efficiently find the k closest data items for any search key. The central insight behind Cubit is to create a keyword metric space that captures the relative similarity of keywords, to assign portions of this space to nodes in a light-weight overlay and to resolve queries by efficiently routing them through this space. The system comprises a protocol for object and node assignment, a gossip-based protocol for maintaining the overlay, and a routing protocol to efficiently route queries. The focus of Cubit is on providing approximate keyword search

---

[*] Dept. of Computer Science, Cornell University, Ithaca NY, 14853.
[†] Microsoft Research, Mountain View, CA 94043.

[1] Optimizations, such as supernodes and expanding ring search, make the broadcast process more efficient, but the primitives are still based fundamentally on flooding.

for multimedia content with limited content description. Keywords are derived from the content's filename and information specific to the content type, such as the comment section of torrent files or the extended video information for YouTube video clips.

An efficient algorithm, based on small-worlds [28], for navigating this keyword metric space enables Cubit to quickly identify approximately matching objects. Cubit assigns a random location in space to each overlay node, and each node maintains the set of objects for which it is the closest. Objects are further replicated to a few closest peers to ensure high availability. Each node keeps track of neighbors in a concentric ring structure based on edit-distance that provides a node with near authoritative information about its local region, and with sufficient amount of out-pointers such that it can forward the query towards more authoritative nodes. Cubit discovers the nodes with keywords that are similar to the target by first examining its local ring members, and retrieving additional candidate nodes from these selected members. These new candidates are closer to the target and have more information in the proximity of the targeted region than the previous node. This protocol quickly converges to the closest nodes with high success rate.

Empirical studies show that search terms typically follow a Zipf [2] rather than a uniform distribution [15], which leads to a naturally skewed load distribution. Consequently, nodes whose IDs lie in the vicinity of popular keywords can become quickly overwhelmed. Traditional load-balancing techniques for DHTs that replicate objects to nearby neighbors cannot be used for approximate matching, as queries cannot be safely short-circuited unless an exact match is found. We introduce a novel load-balancing technique based on virtual nodes to disperse hot-spots in keyword popularity that supports short-circuiting queries for approximate matches.

We evaluate Cubit through both a real deployment in a search plugin for Azureus, a popular BitTorrent client, and large-scale simulations. Cubit outperforms DHT-based approximate search techniques, requiring an order of magnitude fewer RPCs; it can successfully answer 40% more queries than DHTs using Soundex hashing, and can accommodate any language for which a word similarity metric can be defined. Currently, there are more than $6,000$ active users of the Cubit search plugin.

Overall, this paper makes three contributions. First, it describes a *keyword space* that captures the similarity of keywords, and outlines a scalable and efficient protocol for routing queries to nodes that are closest to a search term in the space, thus yielding a DHT with an approximate match primitive. Second, it puts Cubit in context of prior theoretical work on small-world networks, and obtains provable small-world guarantees for the routing protocol which (unlike the notions from prior work) apply to the keyword space. Finally, the paper demonstrates through both a real deployment and large-scale simulations that the system is accurate, efficient, and robust. In particular, it can place the target object in the top 20 results for more than 92% of the queries even with a high degree of perturbation in the search terms.

2 APPROACH

A *keyword* is any word that appears in the title of an object stored in Cubit. In order to fully specify the problem of approximate string matching, we need to choose a notion of distance between two keywords, or more generally between two text strings. Such distance should correspond to our intuition on which strings are similar and which strings are very different. In particular, the distance between a given keyword and its misspelling should be small. Cubit uses the most common notion of distance on strings, the Levenshtein distance, commonly known as the *edit-distance*. It is equal to the minimum number of insertions, deletions and substitutions needed to transform one string to another. The keywords then intrinsically lie in the *keyword space*, a metric space on keywords with a metric given by the edit distance.[3]

Let us consider a typical keyword space taken from the movie database released by NetFlix [4] consisting of about $12,000$ keywords from $17,770$ movie titles. By definition, all edit-distances are integer values. Since most keywords are short, distances in the keyword space tend to be small (see Figure 1). Thus the size of a ball around a typical node grows with the radius much faster than (say) in a two-dimensional grid.

Node ID Assignment. Cubit nodes are distributed in the same space as keywords. Each node in Cubit is assigned a unique string ID chosen from the set of keywords associated with previously inserted objects in the system. The ID of a node determines its "position" in the keyword space. This position determines how a given node is used in Cubit. First, each Cubit node is responsible for storing the set of keywords for which it is the closest node. Second, Cubit implements a distributed protocol which navigates through nodes in the keyword space, gradually zooming in on a neighborhood of a given (possibly misspelled) keyword, and thus locates nodes that store possible matches. The details of the protocol are not critical at this stage; the crucial point is that the navigation happens within the keyword space rather than on a ring or some other highly structured artificial routing space of a typical structured peer-to-peer network.

---

[2] There is also evidence for a flattened Zipf distribution in file-sharing networks [24].

[3] A *metric space* on a set X is a pair (X, σ), where σ is a *metric*, i.e. i.e. a non-negative symmetric function σ that obeys $(\sigma(a,b) = 0 \iff a = b)$ and triangle inequality $\sigma(a,c) \leq \sigma(a,b) + \sigma(b,c)$.

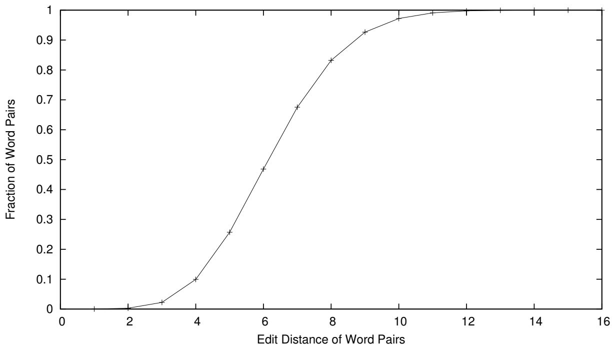


Figure 1: The edit distance between pairs of keywords in the NetFlix data set: most distances are very small.

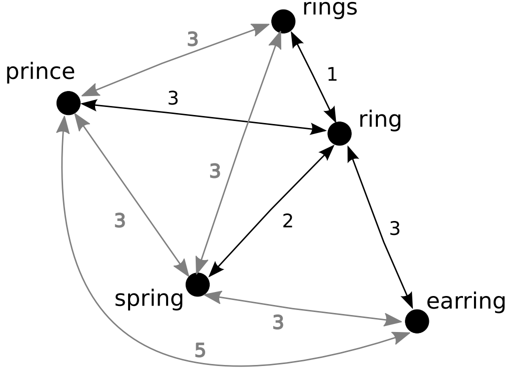


Figure 2: The edit-distance between keywords: a set of five keywords which cannot be embedded into a plane. Once you preserve the distances between four nodes (all but *ring*, the gray edges), the distances to the fifth node are off (the black edges).

Node IDs are chosen to provide a good coverage of the keyword space. A natural approach is to choose node IDs at random. Since the distribution of words in a human language is known to be very different from that of random strings, we choose node IDs at random *among keywords*. Specifically, at join time each node independently selects a random keyword, ensuring uniqueness by detecting ID collisions.

Navigation. The navigation protocol is the core component of Cubit. To support this protocol, Cubit creates and maintains a multi-resolution overlay network on nodes such that each node has several peers at every distance from itself; the peers at a given distance are chosen to maximize the coverage of that region. Such overlay design is inspired by the small-world construction [28, 29] in which a grid is augmented by a sparse set of randomly chosen edges, with roughly the same number of edges for each distance scale. In the resulting graph a simple greedy routing algorithm (which on each step minimizes the distance to target) succeeds in finding short routes to any given target with high probability.

In Cubit, the distance scales are linear rather than exponential because the keyword space has a very small diameter. The small-world-like overlay is created via an underlying low-overhead gossiping protocol under which nodes randomly exchange peer identifiers and thus randomize their peer sets. Since the distance to the target can be easily computed from the corresponding node ID, the greedy routing algorithm requires very little state and is easy to implement in practice. Both the overlay creation and the small-world navigation happen, essentially, in the keyword space. In Section 5 we discuss how the small-world navigation is affected by the properties of this space.

Rejected Alternative: Euclidean embedding. Earlier in this project [48], we advocated representing keywords as points in a low-dimensional Euclidean space (which we term a *hyperspace*). One approach to construct such an embedding is to label each axis of the hyperspace with a string (the anchor points), and define each virtual coordinate of a given keyword as the edit-distance to the corresponding axis label. For instance, for axes *aaa*, *cbc*, *abd*, the keys *abc*, *abd* and *ddd* would map to the points $\langle 2, 1, 1 \rangle$, $\langle 2, 2, 0 \rangle$ and $\langle 3, 3, 2 \rangle$ respectively. This virtual coordinate assignment captures the relative similarities of the strings through the edit-distance to the anchor points. Once nodes and keywords are embedded into a hyperspace, techniques such as CAN [37] and Meridian [47] can be used for navigation in that space.

While this approach gives a clean and intuitively appealing representation of the keyword space, we found that our hyperspace embedding leads to significant embedding errors[4], distorting the navigation. Besides, the navigational framework that we chose to explore does not (really) take advantage of the coordinates. Eventually we found it more fruitful to bypass the embedding and work with the edit distances directly.

To appreciate the difficulty of embedding edit distances into a Euclidean space, consider an example in Figure 2 with a set of five keywords which cannot be embedded into a plane. The embedding becomes increasingly more difficult with additional keywords, even if we allow more dimensions.

3 FRAMEWORK

The basic Cubit routing framework relies on multi-resolution rings to organize peers, a ring membership replacement scheme to maximize the usefulness of ring members, and a gossip protocol for node discovery and membership dissemination. Additionally, the framework

---

[4] That is, the edit distance between two strings may be very different from the corresponding Euclidean distance in the embedding.

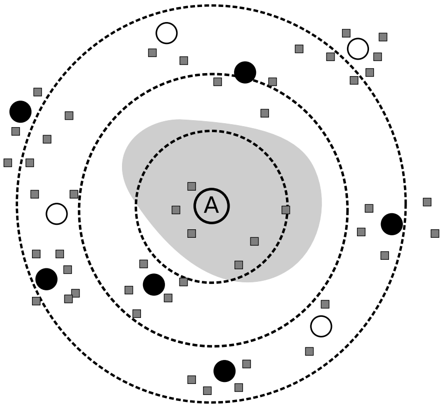


Figure 3: A Cubit node organizes its peers into concentric rings, each with a fixed number of nodes. In this example, the solid circles represent peers in node A's peer-set, the empty circles represent other peers, and the squares represent object keywords in the system. The shaded region depicts the sub-space that is closer to A than any other node. The master record for each keyword in the shaded region is stored at node A.

has mechanisms to proactively maintain object replication for improved resiliency in highly dynamic peer-to-peer systems.

Multi-Resolution Rings. Each Cubit node organizes its peers into a set of concentric rings. In each ring, a node retains a fixed number, k_{ring}, of neighbors whose distance to the host lies within the ring boundaries. This ring structure enables a Cubit node to retain a relatively large number of pointers to other nodes within its vicinity, while also providing a sufficient number of pointers to far-away peers.

The Cubit ring structure is illustrated in Figure 3. The ith ring has inner radius $R_i = \alpha i$ and outer radius R_{i+1}, for $i \geq 0$, where α is a constant. (We use $\alpha = 1$.) Each node keeps track of a finite number of rings; all rings $i > i^*$ for a system-wide constant i^* are collapsed into a single, outermost ring that spans the range $[\alpha i^*, \infty]$.

In addition to the multi-resolution rings, each node maintains a small *leaf set*, a set of nodes used for object replication management and collision detection on node joins. The leaf-set contains a node's (βf_{repl})-closest neighbors, where $\beta \geq 1$ is a parameter and f_{repl} is the *replication factor*; that is, the number of nodes at which each keyword is replicated.

Ring Membership Management. The number of nodes per ring, k_{ring}, represents a trade-off between accuracy and overhead. A large value of k_{ring} allows each node to retain more information for better route selection during query routing, but requires additional overhead in both memory and bandwidth. The utility of a ring member is in relationship to the amount of diversity it can provide to the ring. Diverse ring members provide better coverage and minimize "holes" in the keyword space, reducing the likelihood that a node is overlooked in query routing.

For each ring, the node retains a constant number l_{ring} of additional nodes that serve as potential ring candidates. During ring membership selection, an infrequent periodic event, the node selects a the subset of k_{ring} ring members from the $k_{\text{ring}} + l_{\text{ring}}$ candidates. The goal is to achieve a good coverage of the corresponding annulus in the keyword space. The specific heuristic used to accomplish this is to assign each candidate node a point in the $(k_{\text{ring}} + l_{\text{ring}})$-dimensional space, where each dimension represents its distance to one of the candidate nodes, and choose a subset of k_{ring} nodes that forms a polytope with the largest hypervolume. The quality of the local embedding used in the polytope computation is not critical. Any heuristic for picking a geometrically diverse set of peers would suffice; the polytope volume provides a principled way to select such diverse peers [47].

Gossip Based Node Discovery. A standard anti-entropy push-pull protocol [21] provides node discovery and dissemination between Cubit nodes. At each gossip round, a Cubit node collects a random selection of its ring members, and pushes this collection along with its own node information to a random member in each of its rings. At the same time, it pulls back a random selection of nodes from each of the selected ring members. The exchanged nodes are kept as members in the appropriate ring or as replacement candidates if the ring is full.

Additionally, nodes exchange their leaf-set with their leaf-set members periodically at a more frequent rate, to ensure that changes to the leaf-set are disseminated more quickly than changes to more distant neighbors.

Replication Management. In Cubit, objects are replicated in order to provide high availability. The number of replicas of an object naturally falls over time as nodes exit the system. We introduce a simple replication management protocol to maintain the number of replicas at the desired level f_{repl}.

The *primary node* for a given keyword is the one closest to the keyword, with a fixed tie-breaking rule. This node is responsible for the keyword and its associated objects, and the replication thereof. Each node periodically checks if it is the primary node for the keywords currently at the node. This check can be performed locally by comparing the keywords with the node IDs of the nodes in the leaf-set.[5] Each node ensures that an object is replicated at the $f_{\text{repl}} - 1$ closest leaf-set members for each of its keywords that map to that node. Missing

---

[5] It is possible (though unlikely) that for a brief time interval two or more nodes will consider themselves primary for the same keyword. Such behavior does not reduce accuracy of the search protocol. At worst, it can only *increase* replication level.

Algorithm 1 SEARCH PROTOCOL

```
Require: E: Search event            R: Local ring set
         U: Outstanding queries     H: Leaf set
N ← E.GETREMOTENODE()
I ← E.GETQUERYID()
K ← E.GETFANOUT()
T ← E.GETKEYWORD()
if E.TYPE() = SearchRequest then
    A ← GETKCLOSESTNODES(T, K, R + H)
    N.SEND(SearchReply, I, T, A)
else if E.TYPE() = SearchReply then
    C ← E.GETRESULTS() - CHECKED[I] - PENDING[I]
    CHECKED[I] ← CHECKED[I] + {N}
    PENDING[I] ← PENDING[I] + C - {N}
    A ← CHECKED[I] + PENDING[I]
    A ← GETKCLOSESTNODES(T, K, A)
    if A ⊆ CHECKED[I] then
        for all V in A do
            V.SEND(FetchObjRequest, I, E.SEARCHTERMS())
    else
        for all V in A ∩ C do
            V.SEND(SearchRequest, I, K, D, T)
```

replicas are re-created from the primary copy and disseminated to the appropriate nodes.

4 QUERY ROUTING

The following sections describe protocols that make use of the basic infrastructure described in Section 3 to provide the necessary primitives for performing approximate keyword matching.

4.1 Object Insert

An object in Cubit is fully described by a set of keywords. In the case of our BitTorrent implementation, these keywords are taken from the filename and embedded comments in the torrent file. A copy of the object descriptor is replicated at the r closest nodes to each of its keywords. The form of the object descriptor is unrestricted; in our BitTorrent implementation, a object descriptor is made up of the set of keywords and a pointer to the owner of the torrent file.

When a Cubit node receives an object insertion request, it concurrently issues a closest node search for each keyword using the search protocol described below.

4.2 Search Protocol

The desired property of the search protocol is to obtain the $k*$ objects nearest to the set of keywords, as measured by the phrase distance metric, where $k*$ is a parameter in the system. For each keyword in the search phrase, the protocol obtains the $k*$ closest objects from each node which meets the following *edit distance criterion*: its ID is within an edit-distance of q from the keyword, where q is the product of the keyword length and the expected number of perturbations per character (which is a parameter in the system). The protocol selects n_{min} closest nodes if fewer than n_{min} nodes meet the edit-distance criterion, where n_{min} is called the *search fan-out*.

The protocol runs from a fixed node, called the *local node*. It maintains three lists: the *checked list* of nodes that have already been queried, the *pending list* of nodes waiting to be checked, and the *failed list* of nodes such that the corresponding RPC failed or timed out. Initially all three lists are empty.

The protocol inserts the local node into the pending list and enters the following loop. If there exists a node i in the pending list that meets the edit-distance criterion or is closer to the keyword than the closest n_{min} nodes in the checked list, the local node performs an RPC to node i for some of the members in its ring sets: either for all nodes that meet the the edit-distance criterion or for the l_{min} closest neighbors to the keyword, for some constant $l_{\text{min}} \geq n_{\text{min}}$, whichever is larger. If the RPC fails or times out, node i is moved from the pending list to the failed list. Otherwise, it is relocated to the checked list and the new nodes are placed in the pending list unless they have already been checked or have failed a previous RPC. The loop terminates if such node i does not exist.

The $k*$ closest objects to the set of keywords are retrieved either from all checked nodes that meet the edit-distance criterion, or from the n_{min} closest checked nodes, whichever set is larger. The collected objects for all the search terms are ordered by their phrase distance and the $k*$ closest objects are returned as the result of the search.

Algorithm 1 is the pseudo-code for the search protocol. The edit-distance criterion checks are omitted to improve the clarity and readability of the protocol. Figure 4 illustrates an example search query.

4.3 Node Join

A new node first contacts its given seed nodes to obtain their node IDs and, through a random walk, discovers additional nodes in the network and obtains random keywords from each node. After collecting a sufficient number of nodes, it issues a closest node search for each received keyword. If the closest node's ID is different from the keyword used in the search, then the keyword is used as the node ID for the new node. Simultaneous node joins can, with a very small probability, result in more than one node with the same ID. In this case, the leaf-set discovery will ultimately alert the nodes of the collision, and the node with the lower IP address will drop out and rejoin the system.

Once a unique ID is selected, the new node obtains additional ring members from the ring members of its closest node. It also retrieves the keywords and their associated objects from nodes that are closer to it than the nodes they are currently at. The protocol for this operates iteratively. It asks each of its k closest nodes if there are any objects that should be copied to the new node that it does not already have. If at least one keyword is closer, the protocol repeats with a larger k until

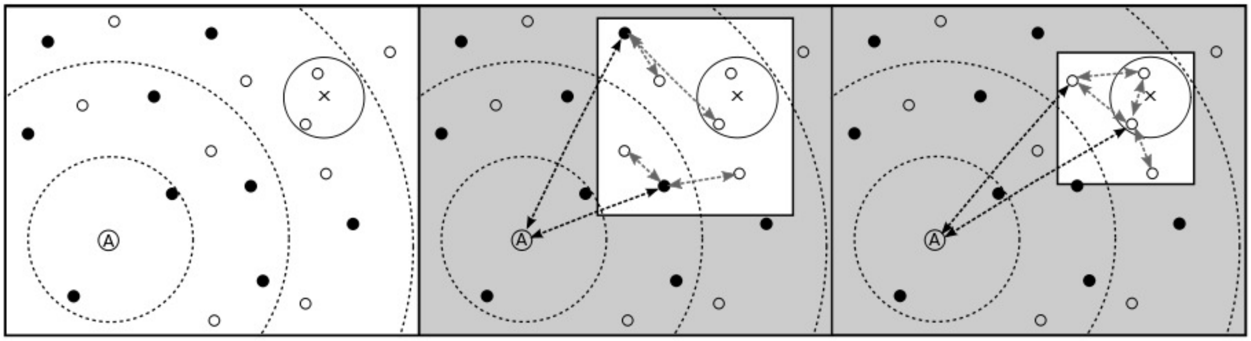


Figure 4: The Cubit search protocol operates iteratively to collect more and more information of the target region. In this example, x is the location of the search term in the keyword space, the solid circles are node A's peers, empty circles are additional nodes in the space, and the circle around x are all nodes within edit-distance q of x. Node A first finds the $n_{\text{min}} = 2$ closest nodes to x from its peer-set, and request their n_{min} closest nodes. In this example, two new closer nodes are discovered and subsequently sent the same query. The protocol terminates when all nodes within the circle around x, or the n_{min} closest nodes have been discovered. These nodes are queried for their closest objects to x.

no new keywords that should be copied are discovered. The new node can optionally, for each object that was copied, request the furthest node with a copy of the object to remove the object from its repository. This can assist the underlying replication management protocol in maintaining the desired replication level.

4.4 Load Balancing

Since search terms tend to follow a Zipf distribution, the resulting skewed load distribution can lead to excess routing load on nodes within the vicinity of popular keywords. Traditional DHT-based load balancing techniques [36, 19, 40] based on object caching by intermediate nodes are not applicable to Cubit, as an intermediate node can not safely short-circuit a search query unless it can find an exact match. We introduce a load-balancing technique that supports short-circuiting of queries for approximate matches.

In Cubit, if the load generated by queries for a popular keyword w overwhelms the available resources of node i, the node can send an off-loading request to its m_{off} closest neighbors (where m_{off} is called the *offload fan-out*) requesting them to create a synthetic node located at w. Nodes receiving such a request create a synthetic node at w whose IP address and port correspond to their own, thus enabling queries for that portion of the keyword space to be terminated at any one of the m_{off} neighbors. The original requester is then tasked with keeping the m_{off} virtual nodes updated with changes to objects in the off-loaded region as well as changes to its leaf-set. If one of the m_{off} nodes becomes overwhelmed, it can request node i to increase the off-loading factor m_{off}. Virtual nodes are not disseminated via gossip and thus do not skew the node distribution. This off-loading operation disperses hot-spots in keyword popularity without requiring global information or coordination. Figure 5 illustrates the protocol.

4.5 Security

A formal treatment of the security properties of a gossip-based small-world network is beyond the scope of this paper. We describe some common attacks targeting the Cubit layer and outline changes to the routing protocol to address them. These changes may incur small performance penalties to query routing.

Keyword Hijacking. An attacker can arbitrarily choose as its node ID a keyword for which it wants to return false information. Such information censorship is possible with unmodified Cubit as the correct execution of the node join protocol cannot be verified by other nodes in the network.

To protect against this attack, Cubit uses a node ID selection protocol that deterministically constructs IDs from the IP address and port of the node. Each Cubit is seeded with the same source of keywords, such as a dictionary, and the hash of the IP address and port is used as an index into the keywords for selecting the node ID. A remote node's ID is verified before it is added into a node's ring set or before it is used in query routing. This modification primarily affects the distribution of objects across the nodes, so the set of seeded keywords should resemble the set of all keywords in the system. The seeded keywords should at least be taken from the same language as the keywords in the system.

Query Disruption. An attacker can try to disrupt query routing by returning false information to the querying node. The disruption can be significant in a localized region, prematurely terminating search and insertion queries. This attack can be circumvented without

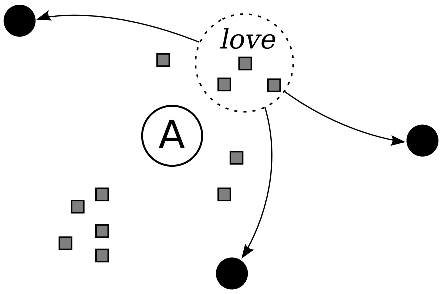


Figure 5: Cubit's load-balancing protocol prevents popular keywords from overwhelming a node. In this example, the keyword "love" is closest to node A and is generating a high degree of load. Node A creates a virtual node centered around the keyword $love$, which includes its leaf set and all objects in the region within p edit-distance from $love$. This virtual node is sent to A's nearest neighbors. Queries that arrive at these neighbors for keywords within an edit-distance p of $love$ can be answered without node A.

changes to the existing query protocol; it can be mostly negated by an increase in the fan-out factor n_{min}. A query only terminates once the top n_{min} nodes to the search term is found. By increasing the n_{min}, an attacker has a proportionally smaller influence on query routing in the region. Queries can typically just route around non-cooperating nodes. Increasing n_{min} comes at a price of additional overhead in query routing. In addition, heavier weight techniques such as PeerReview [25] can be used to identify misbehaving nodes and cleave them from the network.

SPAM Injection. An alternative method to disrupt the system is to increase the noise to signal ratio of the keywords and objects in the system. This attack can be addressed in a number of ways. Cubit can only provide object insert capabilities to trusted users by requiring objects to be signed by a certificate authority. Keyword targeted attacks can be bounded by limiting the injection rate. A node can reject an insert request if the same node has been repeatedly inserting the same or similar keyword. A more complete solution is the introduction of a distributed reputation system [46,20], where poorly rated objects are either discarded or are given a lower rank in response to search queries.

Sybil Attacks. Sybil attacks can be launched against the system, which can allow the attackers to take control of a region of the keyword space. Countermeasures such as [33, 17] can be used to lower the join rate of the attackers, reducing the extent of the attack, or make the attack prohibitively expensive to undertake, though standard impossibility results apply [22].

5 THEORETICAL ANALYSIS

The basic search protocol in Cubit performs a decentralized nearest-neighbor search on the node IDs, using a greedy routing algorithm on the overlay links. In this section we lay out some principled reasons why this protocol works well, i.e. finds near-optimal matches using a small number of hops. The state-of-art theoretical approach to this issue is *small-world networks*, where one investigates whether the routing performance can be guaranteed by randomness and diversity in the overlay.

Let us put Cubit in the context of prior work on small worlds. A typical small-world analysis relies on the properties of the underlying graph or a metric space. The prior work (see [29] for a comprehensive survey) offers small-world constructions for specific graphs such as grids, trees and hypercubes, or "nice" metric spaces such as those with bounded growth, treewidth, grid dimension, or doubling dimension.[6] The provable guarantees tend to be asymptotical, such as $O(\log N)$ hops, where N is the number of nodes. The literature also provides several impossibility results for some seemingly "tractable" metric spaces and "reasonable" overlay constructions [28, 29, 23].

Underlying the small-world overlay in Cubit is the *keyword space* – the metric space on keywords in which distance function is the edit distance. Indeed, the overlay construction in Cubit is tuned to the edit distances between node IDs (which are essentially a random subset of keywords), and the "greedy routing" is greedy with respect to the edit distance between the node ID and the target string. The keyword space is nothing like the spaces considered in prior work on small worlds. Most notably, the distances in the keyword space are small and take a very small number of distinct values (recall Figure 6). Both the small-world-friendly properties from the prior work and the corresponding analyses break simply because of the low maximal to minimal distance ratio.

The goal of this section is to understand small worlds on the keyword space. We ask: **what features of the keyword space make a small-world-type construction possible?** In a more specific sense, we are looking for features that enable a rigorous analysis.

We identify a property of a metric space which is crucial for the algorithm (we call it the *progress ratio*), verify that this property holds on the keyword space, and show that, given a uniform selection of node IDs and of ring members, this property is sufficient to guarantee good performance of the greedy routing. To the best of our knowledge, this property and the corresponding analysis constitute a novel small-world technique.

Setup. For our analysis we consider the basic *greedy*

---

[6] These constructions discuss the *existence* of a suitable overlay, rather than a distributed construction thereof in a peer-to-peer setting.

algorithm: choose any peer which is closer to the target if such peer exists, and stop otherwise. This algorithm completes in a small number of steps (bounded from above by the distance from the original node to the query target) but may stop far from the target. The search protocol used in Cubit builds on this greedy search, but adds more redundancy in order to improve accuracy and thus is likely to work better in practice.

We make the following assumptions about randomness in the overlay (*u.a.r. = uniformly at random*):

(A1) node IDs are distributed *u.a.r.* over Q,

(A2) for each ring i of each node x, the peers are distributed *u.a.r.* over nodes y such that $d(x, y) = i$.

Such assumptions are standard in the small-worlds literature. It suffices to use an approximate *u.a.r.* assumption rather than the exact one.[7]

Let us fix some notation. Let $d(\cdot, \cdot)$ denote the edit distance on strings. Let Q be the set of all keywords. Let Q^* be the set of all queries that we are interested in, e.g. all queries with at most one misspelling. Each Cubit node has an ID in Q. By abuse of notation we extend the edit distance $d(\cdot, \cdot)$ to nodes. For each string w and radius r, the ball *in the keyword space* is denoted $B(w, r) = \{u \in Q : d(u, w) \leq r\}$.

The progress ratio. Following the literature, we'd like to argue that every few hops the search algorithm makes a significant progress towards the target. In prior work, this meant decreasing the distance to target by a constant factor. In our setting it suffices to make *any* progress, i.e., decrease the distance by one.

Consider a query $q \in Q^*$. Let x be the current node, and assume there exist (enough) nodes within distance $r = d(x, q) - 1$ from q. We would like to guarantee that the algorithm can make progress towards q, i.e. that x has a peer in $B(q, r)$. Intuitively, x is likely to have a peer in $B(q, r) \cap B(x, r')$ if the intersection is large compared to both balls. To formalize this intuition, we define a quantity which measures the likelihood of making progress, called the *progress ratio* of pair (x, q):

$$\begin{aligned} \text{ratio}(B, B') &= \frac{|B \cap B'|}{\max(|B|, |B'|)} \\ \textsc{progress}(x, q) &= \max_{r'} \ \text{ratio}(B(x, r'),\, B(q, r)), \\ &\quad\quad \text{where } r = d(x, q) - 1. \end{aligned}$$

Provable guarantees. We formulate a "local" guarantee for a given (x, q) pair, and then a "global" guarantee for the search algorithm. Both guarantees are probabilistic; the probability is over the choice of node IDs and peers. Let k_{ring} be the number of peers per ring.

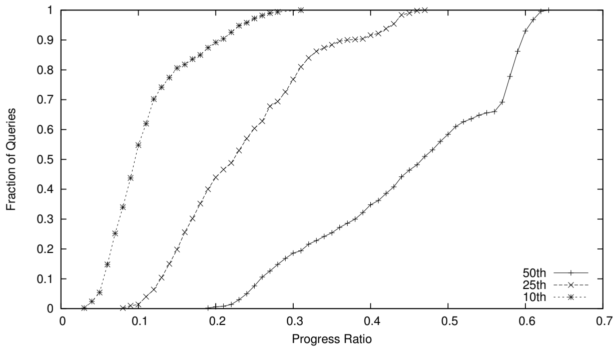


Figure 6: The progress ratios for 1000 randomly chosen node IDs and 500 randomly chosen queries. For each $p = 10, 25, 50$ we present a CDF plot for the p-th percentile progress ratio $r_p(q)$, where the CDF is taken over all queries q. For instance, a high value of $r_{10}(q)$ is a strong positive evidence: namely, for 90% of node IDs the progress ratio is *better* than $r_{10}(q)$.

Lemma 5.1 *Consider an overlay with (A1,A2). Let* $q \in Q^*$ *be a query. Fix node* x *and let* $r = d(x, q) - 1$. *Suppose there are* k *nodes within distance* r *from* q. *Then one of these nodes is a peer of* x *with probability at least* $1 - O(\exp(-\textsc{progress}(x, q) \times \min(k, k_{ring})))$.

Using this lemma, we show that if for a given query $q \in Q^*$ the progress ratio is sufficiently high across all pairs (x, q), then the greedy search algorithm finds a near neighbor with high probability.

Theorem 5.2 *Consider an overlay with (A1,A2). Consider a query* $q \in Q^*$ *such that for some* $k \leq k_{ring}$ *and each node* x *we have* $\textsc{progress}(x, q) \geq \frac{3}{k} \log N$, *where* N *is the number of nodes. Then with probability at least* $1 - O(N^{-2})$ *the greedy search algorithm always finds a* k*-nearest neighbor of* q .

The proofs are relatively straightforward and are omitted from this version due to the space constraints.

Discussion. Our analysis indicates that the progress ratio values on the order of $1/k_{\text{ring}}$ tend to imply good performance of the greedy routing. To verify that the progress ratio values are typically high in the keyword space, we picked 500 queries at random from Q^*, and 1000 node IDs at random from Q, and computed $\textsc{progress}(x, q)$ for every id-query pair (x, q). For a given query q let $r_p(q)$ denote the p-th percentile among the values $\{\textsc{progress}(x, q) : \text{ all nodes } x\}$. In Figure 6 we show how the values $r_p(q)$ are distributed over the queries.

The assumption on the peer distribution provides motivation for the Cubit peer-selection protocol which randomizes and diversifies the peer sets.

---

[7] For instance, one could define: k elements are drawn approximately *u.a.r.* from set S if each element x is drawn independently with probability $p(x) \in (\frac{1}{2n}, \frac{2}{n})$, $n = |S|$.

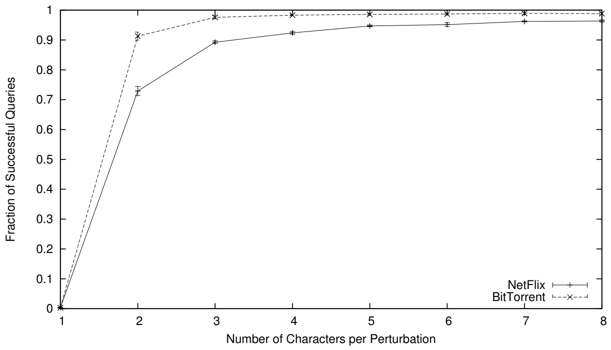
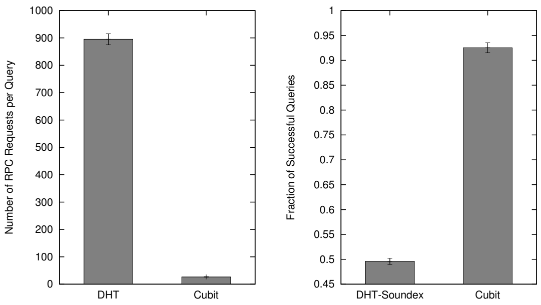


Figure 7: **Number of characters per perturbation (CPP) versus the fraction of successful queries.**



(a)

(b)

Figure 8: (a) **Number of RPC requests per query for a DHT-based system and Cubit.** (b) **Fraction of successful queries for a DHT with Soundex hashing and Cubit.**

6 EVALUATION

We implemented the full protocol described in the preceding section as an Azureus plugin. We evaluate Cubit through both a large-scale simulation on real-world data-sets and a physical deployment on PlanetLab [11].

6.1 Simulation

We use three different real-world data-sets to parameterize our simulations. The first is the NetFlix movie database, consisting of $17,770$ movie titles. We collected our second data-set by crawling a popular BitTorrent website for media files, consisting of over $39,000$ torrents. These two data-sets represent different extremes, with the NetFlix data-set providing clean input with no duplicate entries, in contrast to the much noisier BitTorrent data. Our third data-set is the CiteSeer [2] database with the titles of over $400,000$ academic papers. While not representative of file sharing content, the large data-set enables Cubit's sensitivity to the number of objects in the system to be measured at a much broader scale.

The system is evaluated against search queries constructed from keywords of a randomly chosen title, with perturbations introduced to simulate typos and spelling variations. Only two-thirds of the keywords from each title were used in each search query to closer emulate typical user behavior. The number of characters per perturbation (CPP) parameter is the ratio of a keyword's length to the number of single character faults in the keyword. It is a measure of the signal to noise ratio of search keys and is used to control the difficulty of search queries, where a lower CPP value represents a more difficult query.

In the following experiments, unless specified otherwise, each test consists of 4 runs of 1024 nodes, 10 nodes per ring, a CPP of 4, a search fan-out of 2, a replication factor of 4, with 1000 search queries for each run. The results are presented as the mean result of the runs, and error bars represent 95% confidence intervals. Each simulation run begins from a cold-start, with each new node only knowing at most 8 existing nodes in the network; additional neighbors are discovered through the gossip protocol. An equal fraction of the movies are introduced by each joining node.

We first examine Cubit's accuracy with search queries with increasing levels of difficulty. A search query is considered to be successfully resolved if the original movie it was derived from is a member of the result set, essentially the first page of results presented to the user, which is at most 0.1% of the total number of movies in the system. Figure 7 shows that Cubit can successfully answer queries with three or more characters per perturbation with more than 90% accuracy. Surprisingly, for queries where half the characters in each search keyword are perturbed, Cubit is still able to successfully resolve them more than 75% and 90% of the time for the NetFlix and BitTorrent data-sets respectively. As expected, Cubit's accuracy drops to zero when none of the original characters are kept.

The accuracy metric itself does not capture how much work and how many nodes must be contacted to answer the query. A DHT can be 100% accurate if it searches for every misspelled version of a keyword, but would also be highly inefficient. We illustrate the latent costs in Figure 8(a). We use a basic DHT implementation based on Pastry [41] for comparison, with a base parameter of 16 and a replication factor of 4. The shortest search term is used by the DHT, as it has the fewest error permutations. For search queries where exactly one error is introduced to each keyword, a DHT solution requires nearly 900 RPC requests before finding the sought object. In contrast, Cubit requires only 27 RPC requests, an order of magnitude fewer than the DHT solution, for a query accuracy of more than 96%.

Pairing Soundex hashing, a phonetic algorithm for mapping English words by sound, with DHT routing, as proposed in [49], enables approximate matching without resorting to searching for every possible spelling permutation. Figure 8(b) shows that this approach achieves a success rate below 50% for the sample data used in our experiments.

We next examine the scalability of the Cubit framework. To be able to directly compare experiments with different number of nodes in the network, the number of nodes per ring is configured to be proportional to the logarithm of the system size. Figure 9 shows that increasing system size has a small sub-linear effect on search accuracy. A factor of eight increase in the system size incurs a reduction in accuracy of less than 3%. This stems from a higher node density in the keyword space, which in turn, creates a larger set of equidistant closest nodes to a keyword or a search string. The subset of equidistant nodes discovered in the search determines whether or not the target movie is in the set of results. If this slight loss of accuracy presents a problem, a small increase in the number of nodes per ring or the search fan-out can compensate.

Figure 10 shows that the number of RPC requests per movie and per keyword grows sub-linearly with additional nodes. The RPC requests growth is again due to the larger set of equidistant closest nodes, around the keyword or search string. The growth rate is very low; a factor of eight increase in the system size results in less than a factor of two increase in the number of RPC requests.

Another measure of scalability is Cubit's sensitivity to the number of unique objects in the network. To allow for a more comprehensive evaluation, we use the CiteSeer data-set consisting of more than $400,000$ academic paper titles in our evaluation. In these simulations, rather than returning 0.1% of the total number of unique objects in the system as the result set, we fix the result set to 10 objects to allow for a fair comparison. Figure 11 shows that there is an expected small linear decrease in accuracy with increasing number of objects in the system. A fifty fold increase in objects results in less than 3% decrease in search accuracy. The search accuracy on the CiteSeer data-set is considerably higher than on the NetFlix data-set. This is primarily due to the relatively longer, more distinctive titles found in academic papers, resulting in a sparser, more search friendly keyword space.

The performance of Cubit depends on several key parameters, such as the number of nodes per ring and the query fan-out factor. The number of nodes per ring represents a tradeoff between protocol maintenance bandwidth versus routing accuracy. A low nodes per ring value provides poor coverage of the space and can cause early termination of search queries, where a high nodes per ring value requires additional state to be kept and maintained at each node. Figure 12 shows that accuracy increases dramatically going from two nodes per ring to four, and quickly reaches a plateau at sixteen nodes per ring. The figure also demonstrates that larger systems benefit more from a higher ring size, as additional ring members are necessary to discern distinct regions in the keyspace with increasing node density.

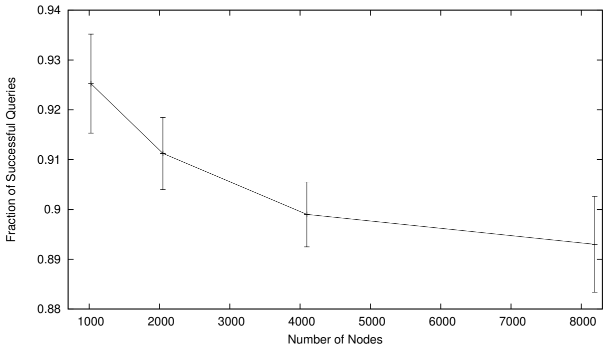


Figure 9: **Number of nodes in the system versus the fraction of successful queries.** Increasing the number of nodes results in a small decrease in search accuracy.

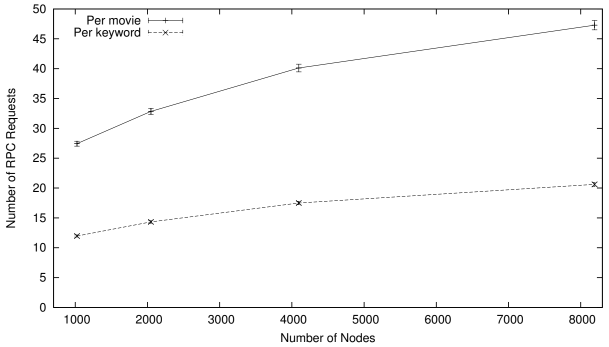


Figure 10: **Number of nodes in the system versus the number of RPC requests.**

The query fan-out bounds the number of closest nodes a query traverses simultaneously, and can significantly improve accuracy by circumventing dead-end paths. For example, a query with a fan-out of two will attempt to find the two closest nodes to the search term at every step, essentially interweaving two simultaneous closest node queries without introducing overlaps in the search space. Figure 13 illustrate that increasing fanout from one to two nets a 8% improvement in accuracy, with further increases netting subsequently smaller gains. However, the accuracy comes at the cost of requiring additional RPC requests. Figure 14 shows that the number of RPC requests increase linearly with the fan-out factor.

The object replication factor also plays a role in the performance of the system. Figure 16 shows that increasing the replication factor from one to four increases

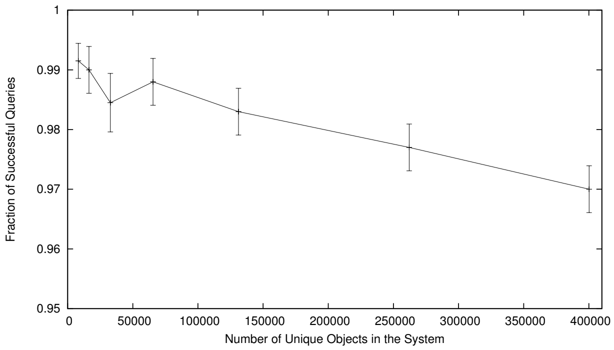


Figure 11: **Number of objects in the system versus the fraction of successful queries.** Simulations performed on the CiteSeer data-set.

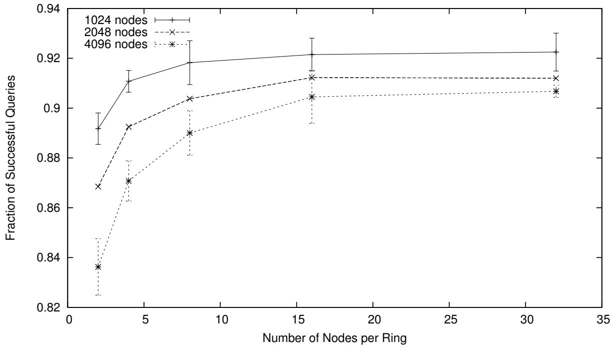


Figure 12: **Number of nodes per ring versus the fraction of successful queries.**

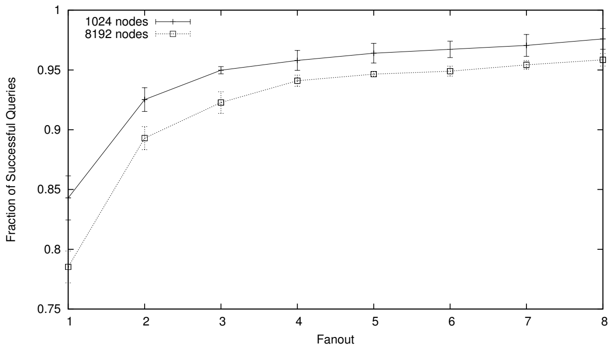


Figure 13: **Search fanout versus the fraction of successful queries.** Increasing search fanout greatly improves search coverage and accuracy.

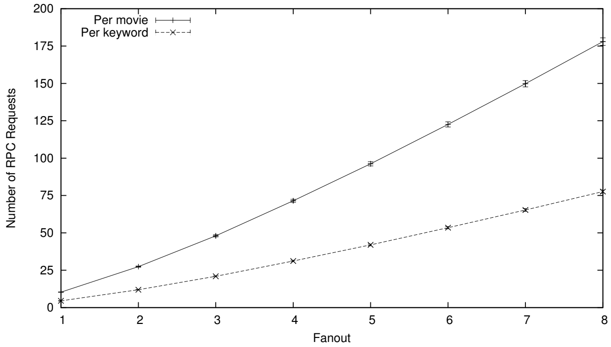


Figure 14: **Search fanout versus the number of RPC requests.** The number of RPC requests per query and per movie both increase linearly with search fan-out.

search accuracy by nearly 20%. Increasing the replication factor beyond four gives only marginal accuracy improvements.

The main tradeoff of a high replication factor is the associated increase in bandwidth for replica management. This bandwidth requirement is proportional to the replication factor, the average number of movies per node, and the node churn rate. To quantify the bandwidth requirement for replica management, we added churn to our simulations. The node lifetime distribution was collected from our Azureus deployment of more than $6,000$ Cubit users. Under this realistic churn scenario, the bandwidth required for replica management is less than 5 KB/s for each Cubit node.

Beyond its effect on maintenance traffic, node churn can also negatively affect search accuracy. This is primarily due to stale ring members that create "holes" in the keyword space, preventing queries from routing to the target region. However, introducing node churn into the simulation results in a barely perceptible decrease in search accuracy (Figure 15). This is because the gossip rate is sufficiently high to detect and remove stale ring members. In our deployment, an average ring member receives a gossip request every two minutes, and the actual measured median lifetime of a node is 20 minutes. Raising the values of other system parameters, such as the number of nodes per ring, query fan-out, and replication factor, provides ways to maintain search accuracy under higher levels of churn.

We next examine how well the load-balancing protocol disperses hotspots in query routing. In this experiment, we overload the system by issuing a misspelled keyword query from 100 randomly selected nodes. In response, the top ten most highly frequented nodes request their neighbors to create virtual nodes. We then repeat the queries and compare the concentration of queries that frequent the top ten most visited nodes before and after virtual node creation. We vary the offload fan-out γ and plot the average number of queries that frequented the top ten nodes and their reduction in average load. Figure 17 shows that the Cubit load-balancing protocol is

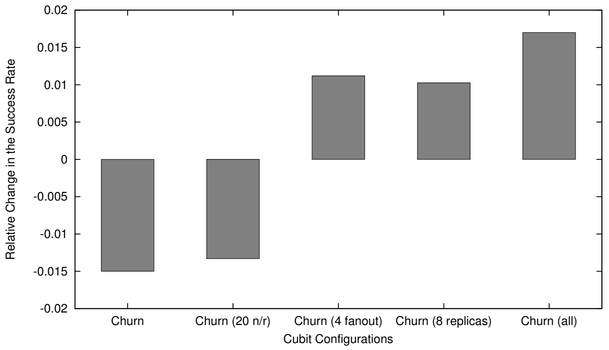


Figure 15: **Relative change in query success rate compared to churn free result.** The first bar shows that realistic churn rates have modest effects. Optimizations, including increased nodes per ring, increased fanout, increased object replication, and all combined, can compensate for the effects of churn.

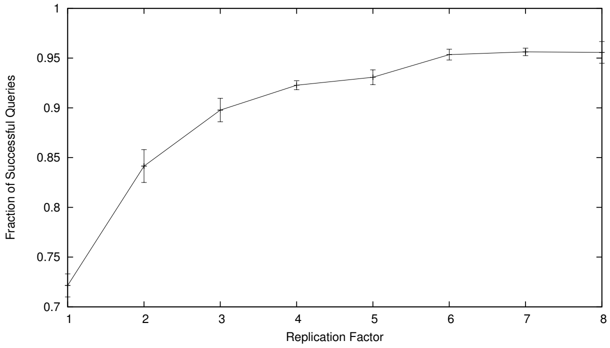


Figure 16: **Replication factor versus the fraction of successful queries.** Modest amounts of replication can yield high recall rates.

effective at reducing the load at request hotspot through the introduction of virtual nodes. Even an off-load fanout of eight is able to reduce the load by more than 40% on average.

6.2 Azureus Deployment

We implemented a Cubit plugin for the Azureus BitTorrent client to provide approximate matching of available torrents. The torrents are currently taken from crawls of popular torrent websites and from the trackerless torrents stored in the Azureus DHT. Torrents in the system automatically expire after a set time-out; persistence beyond a single time-out requires reinjections, similar to OpenDHT [39].

The system is currently deployed, with 107 PlanetLab nodes acting as gateway nodes to the network. More than $10,000$ torrents have been injected into the system, with hundreds of new torrents injected daily. We examine Cubit's accuracy on the Azureus deployment by issuing 125

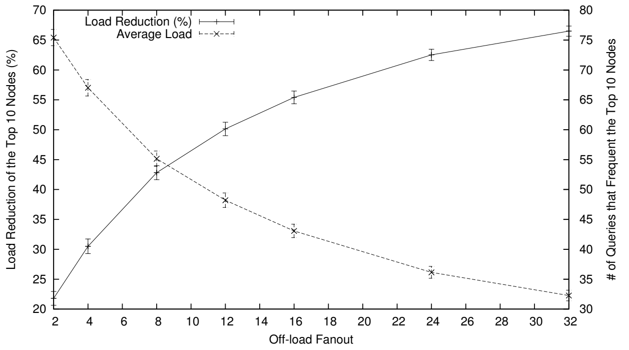


Figure 17: **Offload fanout versus load at hotspots.** Cubit's load balancing protocol is able to significantly spread the load away from load hotspots.

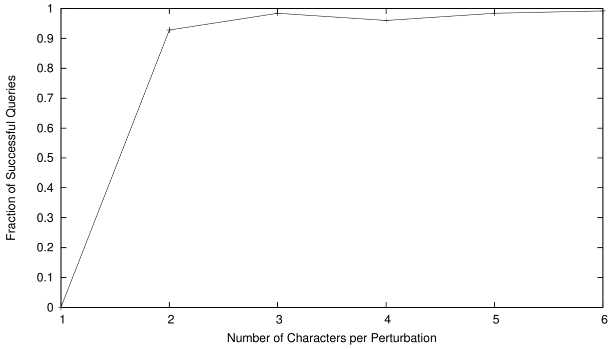


Figure 18: **Number of characters per perturbation (CPP) versus the fraction of successful queries in the Azureus/PlanetLab deployment.**

search queries for each CPP value from one to six. Figure 18 shows that Cubit can successfully answer queries with two or more characters per perturbation with more than 90% accuracy. The small size of the deployment results in better accuracy than predicted by our simulations; we expect a small reduction in accuracy with a larger deployment. There are currently more than $6,000$ active users. The plugin is available at our project website [8].

7 RELATED WORK

Cubit is a loosely structured overlay network that most closely resemble a distributed hash table. It differs from previous DHTs [41,43,50,37,32,27] by providing a novel approximate match primitive rather than supporting only precise lookups.

Query routing in Cubit is similar to routing in CAN [38], SWAM [10], and Meridian [47]. CAN is a coordinate-based approach in which each node knows

---

[8] http://www.cs.cornell.edu/~bwong/Cubit.

its immediate closest neighbor in each of the dimensions and greedily routes to the destination. CAN works best when the embedded node set resembles a grid or a torus; it is not designed to work on highly non-homogeneous point sets such as the (embedded) keyword space. Border cases in dealing with churn makes CAN difficult to implement and deploy in practice. SWAM [10] is similar to CAN but partitions the coordinate space into a Voronoi diagram instead of a regular grid. This provides SWAM with stronger guarantees in performing nearest neighbor search, but incurs additional complexity and overhead to the node join protocol. Meridian is a coordinate-free approach which uses a similar multi-resolution ring structure as Cubit, but targets a very different underlying metric space – that of Internet latencies, which has high diameter and comparatively regular, low-dimensional structure.

Several peer-to-peer systems, e.g. [43, 31, 30], use the overlay routing based on the Small Worlds Networks [28]. These systems use a specific virtual space (e.g. a ring), in which long links are introduced so that a simple greedy routing protocol finds short routes. Inherently, such designs support precise lookups only. A related line of work considers small-world networks on arbitrary underlying spaces, see [29] for a survey. However, this line of work does not tackle the issue of constructing a suitable overlay in a distributed peer-to-peer environment.

Past work has proposed to use the Soundex algorithm to encode keywords by their phonemes before indexing them in a DHT [49]. Unlike edit distance, Soundex is appropriate only for English keywords and is not effective against typing errors.

DPMS [8, 9] provides a less general form of approximate matching suitable only for rearranged substrings. Each document is associated with a set of keywords. Keywords and queries are broken up into fixed size substrings. A query match is found if its substrings are a subset of the document's substrings. The system checks for subset inclusion probabilistically using Bloom filters [14, 16]. The matching primitive in DPMS only accommodates substring matches, does not make a distinction on substring ordering, and it does not find near-matches for queries that are misspelled.

Squid [42] creates a multi-dimensional space using a fixed number of keywords as axes. Each object is represented by a set of keywords, and its position in the multi-dimensional space is based on the prefix match distance between the keywords and the axes. The multi-dimensional space is flattened using space filling curves into a one dimensional space, allowing storage and search to be performed on a DHT. This scheme is primarily targeted at range queries on search terms that are small variations of the axes keywords, rather than for arbitrary search terms.

A number of systems make use of coding techniques to provide approximate search. In P2P-AS [34], an error correcting code is introduced that maps small variations of a keyword into the same hash bin. However, the cost of scaling the number of correctable errors is prohibitive. Another coding based system is LSH Forest [12], which uses locality-sensitive hashing [26] to cluster similar terms. The system is primarily focused on finding similar documents rather than keywords.

pSearch [45,44] uses latent semantic indexing on documents to generate vectors that represent its relative similarity to other documents in the system. CAN [38] is used to traverse this vector space. The focus of pSearch is on finding documents with high semantic relevance to the search keys. It is however unable to match misspelled search keys to documents with correctly spelled keywords, as the search keys and keywords may be typographically similar but are semantically unrelated.

8 CONCLUSION

This paper describes Cubit, a novel approach to efficiently perform approximate matching in peer-to-peer overlays. The key insight behind Cubit is to create a keyword metric space that captures the relative similarity of keywords, to assign portions of this space to nodes in a light-weight overlay and to resolve queries by efficiently routing them through this space, allowing Cubit to quickly identify approximately matching objects to a given set of search terms. The technique is immediately applicable to domains, such as peer-to-peer filesharing, where query terms are provided by users and require a decentralized approximate match against objects in the system.

Cubit has been implemented as a BitTorrent client plugin with more than $6,000$ active users, and evaluated through a PlanetLab deployment as well as through extensive simulations using large, real-world data-sets. The evaluation indicates that Cubit is scalable, accurate, and efficient – it uses an order of magnitude less communication than naive extensions to DHT systems and is nearly twice as accurate as systems based on Soundex hashing. The results show that Cubit can be used to provide approximate matching of keywords. This overall approach may be applicable to other domains where a similarity-based clustering of objects is desired.

References

[1] Britney Spears Spelling Correction. http://www.google.com/jobs/britney.html.

[2] CiteSeer Publication ResearchIndex. http://citeseer.ist.psu.edu/.

[3] Gnutella. http://www.gnutella.com/.

[4] Netflix Prize. http://www.netflixprize.com.

[5] Secrets of the Pirate Bay. http://www.wired.com/science/discoveries/news/2006/08/71543.

[6] The Biggest Ever BitTorrent Leak: MediaDefender Internal Emails Go Public. http://torrentfreak.com/mediadefender-emails-leaked-070915/.

[7] The Pirate Bay. http://thepiratebay.org/blog/68.

[8] R. Ahmed and R. Boutaba. Distributed Pattern Matching for P2P Systems. *NOMS,* Vancouver, 2006.

[9] R. Ahmed and R. Boutaba. Distributed Pattern Matching: A Key to Flexible and Efficient P2P Search. *IEEE Journal on Selected Areas in Communications,* 25(1), 2007.

[10] F. Banaei-Kashani and C. Shahabi. SWAM: A Family of Access Methods for Similarity-Search in Peer-to-Peer Data Networks. *CIKM,* Washington, DC, 2004.

[11] A. Bavier, M. Bowman, B. Chun, D. Culler, S. Karlin, S. Muir, L. Peterson, T. Roscoe, T. Spalink, and M. Wawrzoniak. Operating System Support for Planetary-Scale Network Services. *NSDI,* San Francisco, 2004.

[12] M. Bawa, T. Condie, and P. Ganesan. LSH Forest: Self-Tuning Indexes for Similarity Search. *WWW,* Chiba, 2005.

[13] A. Bharambe, M. Agrawal, and S. Seshan. Mercury: Supporting Scalable Multi-Attribute Range Queries. *SIGCOMM,* Portland, 2004.

[14] B. H. Bloom. Space/time Trade-Offs in Hash Coding with Allowable Errors. *Communications of the ACM,* 13(7), 1970.

[15] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-Like Distributions: Evidence and Implications. *INFOCOM,* New York, 1999.

[16] A. Broder and M. Mitzenmacher. Network Applications of Bloom Filters: A Survey. *Internet Mathematics,* 1(4), 2005.

[17] M. Castro, P. Druschel, A. Ganesh, A. Rowstron, and D. S. Wallach. Secure Routing for Structured Peer-to-Peer Overlay Networks. *OSDI,* Boston, 2002.

[18] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying Peer-to-Peer Networks Using P-Trees. *WebDB,* Paris, 2004.

[19] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage with CFS. *SOSP,* Banff, 2001.

[20] E. Damiani, S. D. C. d. Vimercati, S. Paraboschi, P. Samarati, and F. Violante. A Reputation-Based Approach for Choosing Reliable Resources in Peer-to-Peer Networks. *CCS,* Washington, DC, 2002.

[21] A. Demers, D. Greene, C. Hauser, W. Irish, J. Larson, S. Shenker, H. Sturgis, D. Swinehart, and D. Terry. Epidemic Algorithms for Replicated Database Maintenance. *PODC,* Vancouver, 1987.

[22] J. R. Douceur. The Sybil Attack. *IPTPS Workshop,* Cambridge, 2002.

[23] P. Fraigniaud, E. Lebhar, and Z. Lotker. A Doubling Dimension Threshold $\Theta(\log \log n)$ for Augmented Graph Navigability. *ESA,* pages 376-386, Zürich, 2006.

[24] K. P. Gummadi, R. J. Dunn, S. Saroiu, S. D. Gribble, H. M. Levy, and J. Zahorjan. Measurement, Modeling and Analysis of a Peer-to-Peer File-Sharing Workload. *SOSP,* Bolton Landing, 2003.

[25] A. Haeberlen, P. Kouznetsov, and P. Druschel. PeerReview: Practical Accountability for Distributed Systems. *SOSP,* Stevenson, 2007.

[26] P. Indyk and R. Motwani. Approximate Nearest Neighbor: Towards Removing the Curse of Dimensionality. *STOC,* Dallas, 1998.

[27] F. Kaashoek and D. Karger. Koorde: A Simple Degree-Optimal Distributed Hash Table. *IPTPS Workshop,* Berkeley, 2003.

[28] J. Kleinberg. The Small-World Phenomenon: An Algorithmic Perspective. *STOC,* Portland, 2000.

[29] J. Kleinberg. Complex Networks and Decentralized Search Algorithms. *Intl. Congress of Mathematicians,* 2006.

[30] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy: A Scalable and Dynamic Emulation of the Butterfly. *PODC,* Monterey, 2002.

[31] G. Manku, M. Bawa, and P. Raghavan. Symphony: Distributed Hashing in a Small World. *USITS,* Seattle, 2003.

[32] P. Maymounkov and D. Mazieres. Kademlia: A Peer-to-Peer Information System Based on the XOR Metric. *IPTPS Workshop,* Cambridge, 2002.

[33] R. C. Merkle. Secure Communications Over Insecure Channels. *Communications of the ACM,* 1978.

[34] A. Mowat, R. Schmidt, M. Schumacher, and I. Constantinescu. Extending Peer-to-Peer Networks for Approximate Search. *SAC,* Fortaleza, 2008.

[35] A. Parker. P2P in 2005. 2006. CacheLogic presentation.

[36] V. Ramasubramanian and E. G. Sirer. Beehive: O(1) Lookup Performance for Power-Law Query Distributions in Peer-to-Peer Overlays. *NSDI,* San Francisco, 2004.

[37] S. Ratnasamy, P. Francis, M. Hadley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. *SIGCOMM,* San Diego, 2001.

[38] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A Scalable Content-Addressable Network. *SIGCOMM,* San Diego, 2001.

[39] S. Rhea, B. Godfrey, B. Karp, J. Kubiatowicz, S. Ratnasamy, S. Shenker, I. Stoica, and H. Yu. OpenDHT: A Public DHT Service and Its Uses. *SIGCOMM,* Philadelphia, 2005.

[40] A. Rowstron and P. Druschel. Storage Management and Caching in PAST, a Large-Scale, Persistent Peer-to-Peer Storage Utility. *SOSP,* Banff, 2001.

[41] A. Rowstron and P. Druschel. Pastry: Scalable, Distributed Object Location and Routing for Large-Scale Peer-to-Peer Systems. *Middleware,* Heidelberg, 2001.

[42] C. Schmidt and M. Parashar. Flexible Information Discovery in Decentralized Distributed Systems. *HPDC,* Seattle, 2003.

[43] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. *SIGCOMM,* San Diego, 2001.

[44] C. Tang, S. Dwarkadas, and Z. Xu. On Scaling Latent Semantic Indexing for Large Peer-to-Peer Systems. *SIGIR,* Sheffield, 2004.

[45] C. Tang, Z. Xu, and S. Dwarkadas. Peer-to-Peer Information Retrieval Using Self-Organizing Semantic Overlay Networks. *SIGCOMM,* Karlsruhe, 2003.

[46] K. Walsh and E. G. Sirer. Experience with a Distributed Object Reputation System for Peer-to-Peer Filesharing. *NSDI,* San Jose, 2006.

[47] B. Wong, A. Slivkins, and E. G. Sirer. Meridian: A Lightweight Network Location Service Without Virtual Coordinates. *SIGCOMM,* Philadelphia, 2005.

[48] B. Wong, Y. Vigfússon, and E. G. Sirer. Hyperspaces for Object Clustering and Approximate Matching in Peer-to-Peer Overlays. *HotOS Workshop,* 2007.

[49] M.A. Zaharia, A. Chandel, S. Saroiu, and S. Keshav. Finding Content in File-Sharing Networks When You Can't Even Spell. *Intl. Workshop on P2P Systems,* Bellevue, 2007.

[50] B. Zhao, J. Kubiatowicz, and A. Joseph. Tapestry: An Infrastructure for Fault-Tolerant Wide-Area Location and Routing. UC Berkeley, Technical Report UCB/CSD-01-1141, Berkeley, 2001.