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 [31]. 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 [2] 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 [37, 39, 46, 33, 28, 23] 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* [10, 15] 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.

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

[*]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.

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.

An efficient algorithm, based on small-worlds [24], 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 [12], 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.

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 obtains provable guarantees on the performance of this protocol, using a novel small-world technique which, unlike the notions from prior work, applies 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]There is also evidence for a flattened Zipf distribution in file-sharing networks [21].

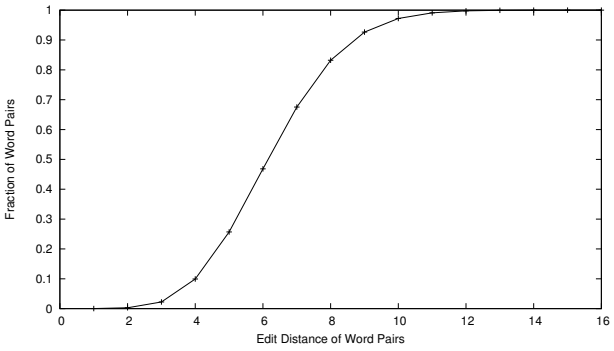

Figure 1: The edit distance between pairs of keywords in the Netflix data set: most distances are very small.

The rest of the paper is structured as follows. Section 2 describes the keyword space and Cubit's general approach to provide an approximate match primitive. Section 3 defines Cubit's routing framework, and Section 4 specifies the query routing protocols that make use of the framework. Section 5 provides a theoretical analysis of the search algorithm in Cubit, proving that it can find the near neighbors to a search term with high probability. Section 6 evaluates the accuracy and performance of Cubit, Section 7 discusses related work and Section 8 summarizes our contributions.

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.[3] 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 induced on keywords by the edit distance.

[3]In principle we could use any such distance as long as it is a *metric*, i.e. a non-negative symmetric function σ that obeys $(\sigma(a,b) = 0 \iff a = b)$ and triangle inequality $\sigma(a,c) \leq \sigma(a,b) + \sigma(b,c)$.

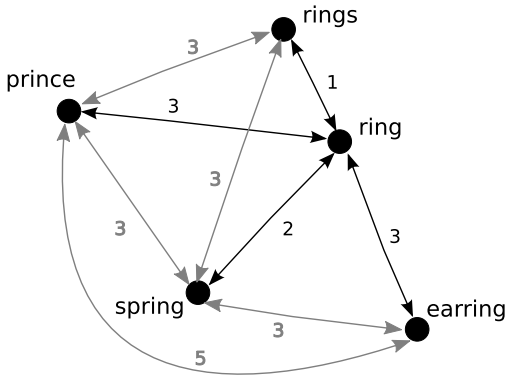

Figure 2: The edit-distance between keywords: a set of five keywords which cannot be embedded into a plane.

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). This implies that, for any keyword u, the number of keywords within edit-distance r from u grows very fast with r, so the keyword space is very different from a low-dimensional grid. Moreover, it is a known theoretical fact that a low diameter metric space is very different from *any* point set in a low-dimensional Euclidean space. 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.[4]

Phrase Matching. Search queries typically consist of more than one search term. For example, a user may search for a long movie title using only a misspelled subset of the many keywords in the original title. A phrase distance that closely tracks the user's intent, as a function of the search terms and the object keywords, is not as well defined as it is for single keywords. In Cubit, we introduce a simple phrase distance metric which we call *Additive Minimum Edit-Distance* (AMED). In AMED, the distance is the sum of the minimum edit-distance of each search term across the set of object keywords.

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.

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 [24, 25] 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.

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: Hyperspace embedding. In previous work [44], we advocated representing keywords as points in a low-dimensional Euclidean space, termed a *hyperspace*. One approach to achieve such an embedding is to label each axis of the hyperspace with a string, and define each virtual coordinate of a given key-

[4] While we did not investigate whether the keyword space is close to a point set in a *high-dimensional* Euclidean space (more precisely, d-dimensional for some $d \ll \#keywords$), even such a weak property seems unlikely due to the highly irregular nature of edit distances.

word 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 string labels. In essence, axis labels act as anchor points, and each component of an object's coordinate provides the distance of the object from that anchor point. Much like the Post Office metric on normed vector space [3], the distance from each anchor point clusters similar objects to the extent differentiable by that axis label, assigning them similar coordinates. The intuition is that similar strings will have similar edit-distance to the corresponding axis labels, especially if the axis labels are well-chosen, for instance, by randomly selecting from the keywords themselves.

Once nodes and keywords are embedded into a hyperspace, a number of different techniques can be used to navigate through the space. CAN [33] can find the closest node to a coordinate in $d\,N^{1/d}$ hops (where d is the dimension) given a uniform distribution of nodes in the space. We previously examined the feasibility of a design based on Meridian [43] that is light-weight and achieves $\log N$ hop routing with high probability given similar assumptions.

However, a hyperspace embedding introduces intrinsic embedding errors, as the keyword space can not be perfectly mapped into even a high-dimensional Euclidean space. Moreover, increasing the dimensionality can improve the distinguishing power of the embedding, but may also cause overfitting and degrade performance. While even a distorted embedding can potentially result in a metric space that provides good recall accuracy and simplifies routing, we observed that Cubit achieves much higher accuracy *without* the embedding. In this paper, we bypass the embedding and present a modified approach that routes within the keyword space directly without computing coordinates for nodes or objects.

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

3.1 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.

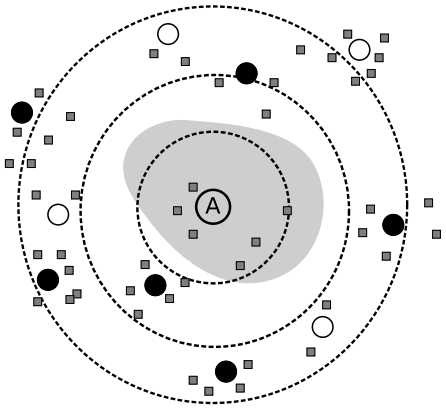

Figure 3: A Cubit node organizes its peers into a set of concentric rings, each with a fixed number of nodes. In this example, the solid circles represent peers in node A's peer-set, the empty circles represent other peers, and the squares represent object keywords in the system. The shaded region depicts 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.

The Cubit ring structure is illustrated in Figure 3. The ith ring has inner radius $r_i = \alpha i$ and outer radius $R_i = \alpha(i+1)$, for $i \geq 0$, where α is a constant that determines ring-width. 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.

3.2 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}

Algorithm 1 MAINTENANCE PROTOCOL

```
          E: Timeout event    R: Local ring set
Require:  L: Local node       O: Object repository
          H: Leaf set         Y: Replication factor
 1: if E.TYPE() = GossipTimer then
 2:   W ← R.SELECTRANDOMNODES()
 3:   for all N in W do
 4:     N.SEND(GossipRequest, (W - N + L))
 5:   W ← H.GETNODES()
 6:   for all N in H.GETNODES() do
 7:     N.SEND(GossipRequest, (H.GETNODES() - N + L))
 8: else if E.TYPE() = ReplacementTimer then
 9:   D ← R.GETRANDOMRINGINDEX()
10:   A ← R.GETPRIMARY(D) + R.GETSECONDARY(D)
11:   B ← {}
12:   while A.LENGTH() > R.MAXNODESPERRING() do
13:     M, V ← NIL, 0
14:     for all N in A do
15:       S ← POLYTOPEVOLUME(A - N)
16:       if M = NIL or S > V then
17:         M, V ← N, S
18:     A, B ← A - M, B + M
19:   R.SETRING(D, A, B) {Set A to primary, B to secondary}
20: else if E.TYPE() = ReplicaTimer then
21:   M ← O.GETALLMASTERREPLICAS(H.GETNODES())
22:   for all C in M do
23:     for all N in H.GETCLOSEST(Y-1, C) do
24:       N.SEND(CheckKeyRequest, C)
```

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 [43].

3.3 Gossip Based Node Discovery

A standard anti-entropy push-pull protocol [18] 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.

3.4 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 replicas are re-created from the primary copy and disseminated to the appropriate nodes. Algorithm 1 illustrates Cubit's periodic maintenance operations.

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, we cache torrent files wholesale, which are typically small and uniform in size. For large objects, the overlay could be extended to cache a pointer to the owner of the actual object.

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_{closest} objects to the set of keywords, as measured by the phrase distance metric, where k_{closest} is a parameter in the system. For each keyword in the search phrase, the protocol obtains the k_{closest} closest objects from each

[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 2 SEARCH PROTOCOL

Require: E: Search event R: Local ring set
U: Outstanding queries H: Leaf set

```
 1: N ← E.GETREMOTENODE()
 2: K ← E.GETFANOUT()
 3: D ← E.GETDISTANCEBOUND()
 4: T ← E.GETKEYWORD()
 5: if E.TYPE() = SearchRequest then
 6:    A ← GETNODESWITHINBOUND(T, D, R + H)
 7:    if A.LENGTH() < K then
 8:       A ← GETKCLOSESTNODES(T, K, R + H)
 9:    N.SEND(SearchReply, E.QID(), T, A)
10: else if E.TYPE() = SearchReply then
11:    B ← S.GETSEARCHQUERY(I)
12:    B.CHECKED = B.CHECKED + {N}
13:    B.PENDING = B.PENDING - {N}
14:    if DISTANCE(N, T) ≤ D then
15:       N.SEND(FetchObjRequest, E.QID(), B.SEARCHTERMS())
16:       B.FETCHED = B.FETCHED + {N}
17:    for all V in E.GETCLOSEST()- B.CHECKED do
18:       B.PENDING = B.PENDING + {V}
19:    A ← B.CHECKED + B.PENDING
20:    A ← GETKCLOSESTNODES(T, K, A)
21:    if A ⊆ B.CHECKED then
22:       for all V in A - B.FETCHED do
23:          V.SEND(FetchObjRequest, E.QID(), B.SEARCHTERMS())
24:          B.FETCHED = B.FETCHED + {V}
25:    else
26:       for all V in A ∩ B.PENDING do
27:          V.SEND(SearchRequest, E.QID(), K, D, T)
```

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 $l_{\min} \geq n_{\min}$, 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} 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} closest objects are returned as the result of the search.

Algorithm 2 is the pseudo-code for the search protocol, and 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 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 [32, 16, 36] 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-*

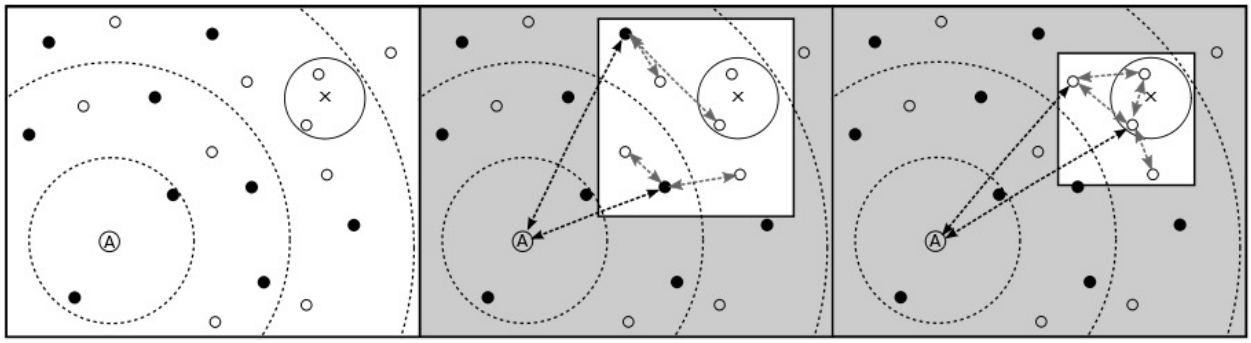

Figure 4: The Cubit search protocol operates iteratively to collect more and more information of the target region. In this example, x is the location of the search term in the keyword space, the solid circles are node A's peers, empty circles are additional nodes in the space, and the circle around 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.

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

For deployments in adversarial environments, small changes to the Cubit query routing protocol are necessary to protect the system against attacks on overlay networks. 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 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.

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 [42,17], where poorly rated objects are either discarded or are given a lower rank in response to search queries.

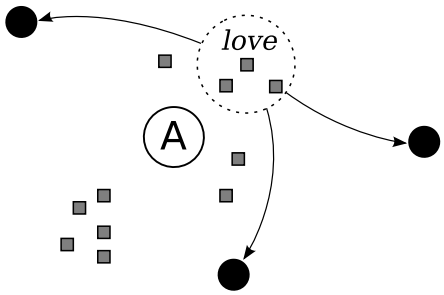

Figure 5: Cubit's load-balancing protocol prevents popular keywords from overwhelming a node. In this example, the keyword "love" is closest to node A and is generating a high degree of load. Node A creates a virtual node centered around the keyword $love$, which includes its leaf set and all objects 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.

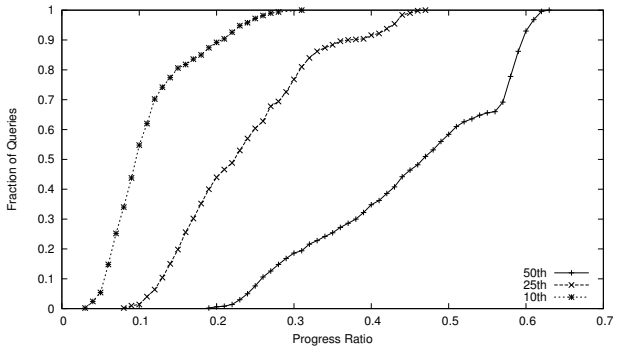

Figure 6: The progress ratios for 1000 randomly chosen node IDs and 500 randomly chosen queries. For each $p = 10, 25, 50$ we present a CDF plot for the p-th percentile progress ratio $r_p(q)$, where the CDF is taken over all queries q. For instance, a high value of $r_{10}(q)$ is a strong positive evidence: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

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 [29, 14] 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 [19].

5 THEORETICAL ANALYSIS

The basic search protocol in Cubit performs a decentralized nearest-neighbor search on the node IDs. 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.

We focus on the underlying *keyword space* – the metric space induced by the edit distance on the set of keywords. Cubit constructs a small-world-type overlay network on node IDs, which are essentially a random subset of the keyword space, and uses greedy search algorithm on the overlay links. The existing theoretical constructions of small-world overlays (see [25] for a comprehensive survey) either assume a specific underlying graph (e.g. a grid, a tree, or a hypercube), or rely on "nice" features of the underlying metric space, such as bounded growth, treewidth, grid dimension, or doubling dimension.[6] Moreover, the literature provides several impossibility results for some seemingly "tractable" metric spaces and "reasonable" overlay constructions [24, 25, 20]. Therefore we ask: **what are the features of the keyword space that make a small-world-type construction possible?**

The techniques from prior work on small worlds do not help us answer this question because the keyword space is nothing like the spaces considered in prior work. Both the small-world-friendly properties and the corresponding analysis break due to the fact that the distances in the keyword space are small (recall Figure 6). Moreover, the distances in the keyword space take a very small number of distinct values.

We develop a new small-world technique by identifying a metric property which is crucial for the algorithm, and verify that this property holds on the keyword space. We show that, given a uniform selection of node IDs and of ring members, this property is sufficient to guarantee good performance. To the best of our knowledge, this property has not appeared in the literature.

Preliminaries. Let $d(\cdot, \cdot)$ denote the edit distance on strings. Let Q be the set of all keywords. For each string w and radius r, the ball *in the keyword space* is denoted

$$B(w, r) = \{u \in Q : d(u, w) \leq r\}.$$

Suppose we are interested in queries with at most 1 misspelling. Then the set of all possible queries is

$$Q^* = \{w \in W : d(w, u) \leq 1 \text{ for some } u \in Q\}.$$

Each Cubit node has an ID in Q. By abuse of notation we extend the edit distance $d(\cdot, \cdot)$ to nodes. Let N be the number of nodes in the system, and let k_{ring} be the number of peers per ring.

[6] Note that most of these constructions assume the *existence* of a suitable overlay, rather than provide a distributed construction thereof in a peer-to-peer setting.

The progress ratio. While in prior work search algorithms need to cut the distance to target by a constant factor in each "phase", in our setting it suffices to make *any* progress (i.e., decrease the distance by one).

Consider a query $q \in Q^*$, and let x be the current node. Suppose there exist nodes within distance $r = d(x,q) - 1$ from q. If one of these nodes is a peer of x, then the search algorithm can make progress towards q. Intuitively, such a peer is likely to exist within distance r' from x if the intersection of $B(x,r')$ and $B(q,r)$ is large compared to both balls.[7] Therefore, let us define a quantity which measures the likelihood of making progress, called the *progress ratio* of pair (x,q):

$$\text{ratio}(B,B') = \frac{|B \cap B'|}{\max(|B|,|B'|)}$$

$$\text{PROGRESS}(x,q) = \max_{r'} \text{ ratio}(B(x,r'),\, B(q,r)) \quad \text{where } r = d(x,q) - 1.$$

Provable guarantees. To capture the above intuition in a rigorous form, we need to make explicit some assumptions about the selection of node IDs and peers. Let us say that Cubit is *well-formed* if node IDs are distributed uniformly at random over Q, and for each ring-i peers of node x are distributed uniformly at random over the nodes y such that $d(x,y) = i$.[8]

The guarantee for a given (x,q) pair can be formulated as follows:

Lemma 5.1 *Suppose Cubit is well-formed. Consider a query $q \in Q^*$. 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*[9] *at least* $1 - O(\exp(-\text{PROGRESS}(x,q) \times \min(k, k_{ring})))$.

Let us use this lemma to derive a "global" guarantee for the search algorithm. We will consider a *greedy* search algorithm which, at every step, forwards the query to any peer which is closer to the target if such peer exists, and stops 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 the greedy search, but adds more redundancy in order to improve accuracy and thus is likely to work better in practice.

We show that for a given query $q \in Q^*$ such that the progress ratio is sufficiently high across all pairs (x,q), the greedy search algorithm finds a k-nearest neighbor with high probability.

Theorem 5.2 *Suppose Cubit is well-formed. Consider a query $q \in Q^*$ such that for some $k \le k_{ring}$ and each node x we have* $\text{PROGRESS}(x,q) \ge \frac{3}{k} \log N$. *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. The take-away from our analysis is that the progress ratio values on the order of $1/k_{\text{ring}}$ tend to imply good performance. We verified that the progress ratio values are typically high in the keyword space. To this end, we picked 500 queries at random from Q^*, and 1000 node IDs at random from Q. We computed $\text{PROGRESS}(x,q)$ for every id-query pair (x,q). To represent these findings, for every fixed query q let $r_p(q)$ denote the p-th percentile progress ratio for q, that is the p-th percentile among the values $\text{PROGRESS}(x,q)$. 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.

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 datasets and a physical deployment on PlanetLab [8].

6.1 Simulation

We use two 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. The two data-sets represent different extremes, with the Netflix dataset providing clean input with no duplicate entries, in contrast to the much noisier BitTorrent data.

Each search query was constructed from keywords of a randomly chosen movie title, with perturbations introduced to simulate typos and spelling variations. Only two-thirds of the keywords from the movie title were used in each search query to closer emulate typical user behavior. The number of characters per perturbation parameter (CPP) is used to control the difficulty of a search query, 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 search fan-out of 2, a replication factor of

[7] We compare the intersection with $B(q,r)$ in order to argue that the former is likely to contain some node IDs as long as the latter does, regardless of the number of nodes in the system.

[8] Assuming "... such that $d(x,y) \le i$" would work, too.

[9] Here and in Theorem 5.2, the probability is over the choice of node IDs and peers; the search algorithm is deterministic.

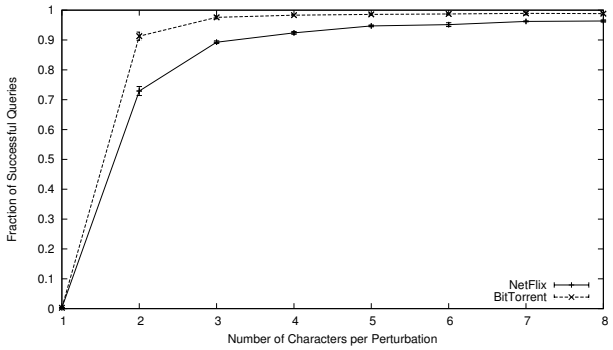

Figure 7: **Number of characters per perturbation (CPP) versus the fraction of successful queries.**

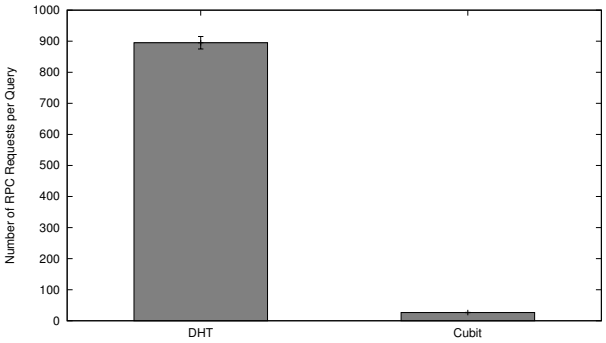

Figure 8: **Number of RPC requests per query for a DHT-based system and Cubit.**

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 are incorrectly spelled, Cubit is still able to successfully resolve them more then 75% and 90% of the time for the Netflix and BitTorrent data-sets respectively. As expected, Cubit's accuracy drops to zero when every character is incorrectly spelled.

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. We use a basic DHT implementation based on Pastry [37] 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 [45], enables approximate matching without resorting to searching for every possible spelling permutation. Figure 9 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 10 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 11 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.

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 determines the amount of information a node has of other nodes in the keyword space. A low nodes per ring value

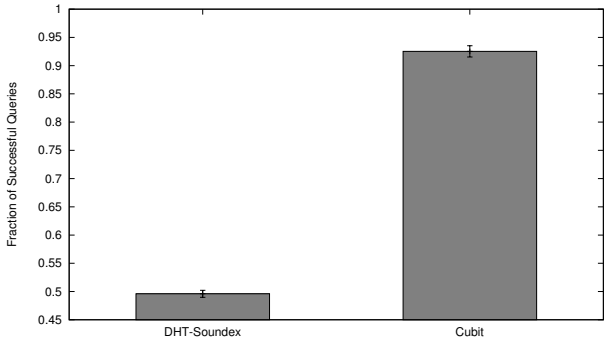

Figure 9: **Fraction of successful queries for a DHT with Soundex hashing and Cubit.**

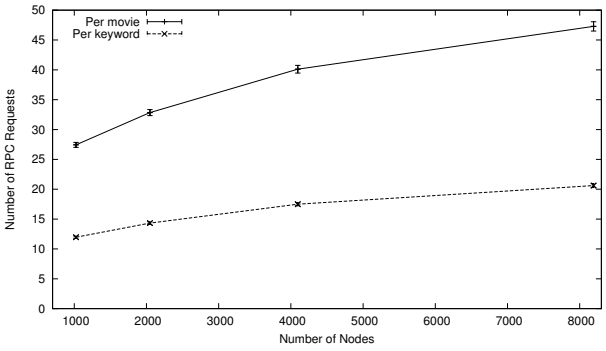

Figure 11: **Number of nodes in the system versus the number of RPC requests.**

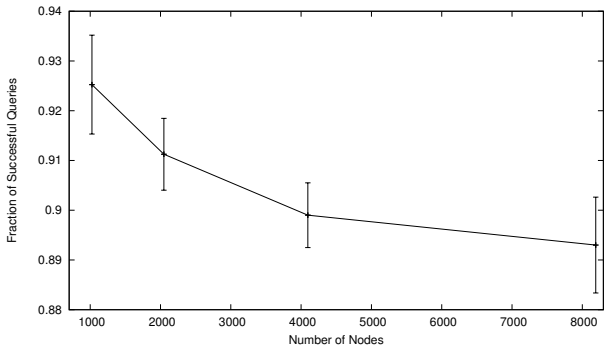

Figure 10: **Number of nodes in the system versus the fraction of successful queries.** Increasing the number of nodes results in a small sub-linear decrease in search accuracy.

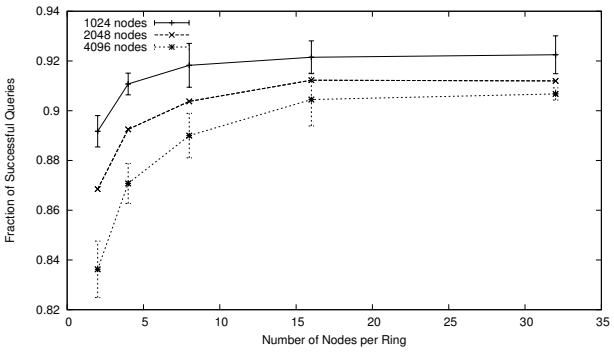

Figure 12: **Number of nodes per ring versus the fraction of successful queries.** The number of nodes per ring affects a node's coverage of the keyword space.

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.

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.

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 15 shows that the Cubit load-balancing protocol is 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.

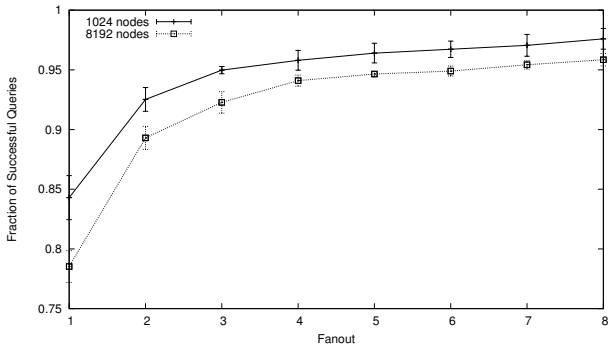

Figure 13: **Search fanout versus the fraction of successful queries.** Increasing search fanout, which dictates the number of top candidates to traverse simultaneously, greatly improves search coverage and accuracy.

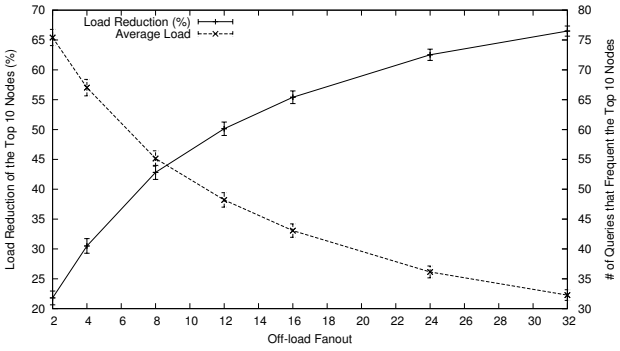

Figure 15: **Offload fanout versus load at hotspots.** Cubit's load balancing protocol is able to significantly spread the load away from load hotspots.

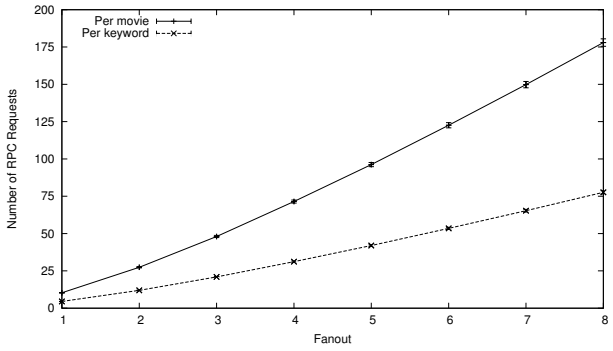

Figure 14: **Search fanout versus the number of RPC requests.** The number of RPC requests per query and per movie both increase linearly with search fan-out.

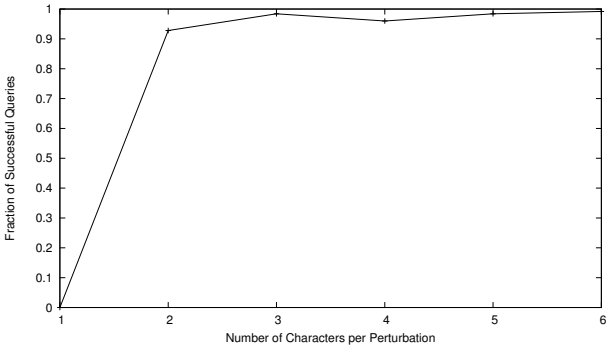

Figure 16: **Number of characters per perturbation (CPP) versus the fraction of successful queries in the Azureus/PlanetLab deployment.**

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 [35].

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 search queries for each CPP value from one to six. Figure 16 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. The plugin is available at our project website [10].

7 RELATED WORK

Cubit is a loosely structured overlay network that most closely resemble a distributed hash table. It differs from previous DHTs [37,39,46,33,28,23] by providing a novel approximate match primitive rather than supporting only precise lookups.

Query routing in Cubit is similar to routing in CAN [34], SWAM [7], and Meridian [43]. CAN is a coordinate-based approach in which each node knows 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-homogenous

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

point sets such as the (embedded) keyword space. Border cases in dealing with churn makes CAN difficult to implement and deploy in practice. SWAM [7] 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. [39, 27, 26], use the overlay routing based on the Small Worlds Networks [24]. 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 [25] 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 [45]. Unlike edit distance, Soundex is appropriate only for English keywords and is not effective against typing errors.

DPMS [5, 6] 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 [11, 13]. 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 [38] 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 [30], 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 [9], which uses locality-sensitive hashing [22] to cluster similar terms. The system is primarily focused on finding similar documents rather than keywords.

pSearch [41, 40] uses latent semantic indexing on documents to generate vectors that represent its relative similarity to other documents in the system. CAN [34] is used to traverse this vector space. The computational overhead in using latent semantic indexing is significantly more than edit-distance computations, and the high dimensionality vector spaces created by latent semantic indexing requires a large amount of state to be maintained per CAN node.

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, 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.

9 ACKNOWLEDGMENTS

This work was supported in part by NSF-TRUST 0424422 and NSF-CAREER 0546568 grants.

References

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

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

[3] Metric Space. http://en.wikipedia.org/wiki/Metric_space.

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

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

[6] 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.

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

[8] 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, California, March 2004.

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

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

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

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

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

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

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

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

[17] 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, November 2002.

[18] 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, Canada, August 1987.

[19] J. R. Douceur. The Sybil Attack. *IPTPS Workshop,* Cambridge, Massachusetts, March 2002.

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

[21] 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, New York, October 2003.

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

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

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

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

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

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

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

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

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

[31] A. Parker. P2P in 2005. January 2006. CacheLogic presentation.

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

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

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

[35] 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, Pennsylvania, September 2005.

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

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

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

[39] 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, California, August 2001.

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

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

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

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

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

[45] 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, Washington, February 2007.

[46] 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, California, April 2001.