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. This overlay also enables multi-keyword searches and supports boolean expressions over keywords. 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 [34]. 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 Gnutella [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 [39, 41, 47, 36, 29, 24] 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* [14, 17] 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, the world's largest torrent aggregator has been the target of attacks by hackers [3], was forced to shut down temporarily [4], and may be forced to shut down permanently [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 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 infor-

*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.

mation for YouTube video clips.

An efficient algorithm, based on small-worlds networks [26], 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 out-pointers to forward queries towards more authoritative nodes. Cubit discovers the nodes with keywords that are similar to the search term by first examining its local ring members, and retrieving additional closer candidate nodes from these selected members. The search is repeated with these new candidates that have more information in the proximity of the search term's region than the previous members. This protocol quickly converges to the closest nodes with a high success rate.

Search queries in Cubit are not limited to single keywords. The search primitive supports using a multi-word similarity metric to match search queries to objects. Cubit also supports combining keywords using user-specified boolean operators, enabling users to precisely define their search query. This is done efficiently by leveraging probabilistic data structures to represent, disseminate, and operate on relatively large intermediate result sets with low, constant size overheads.

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 30% 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 94% of the queries even with a high degree of perturbation in the search terms.

2 APPROACH

An object stored in Cubit is characterized by one or more *keywords*. Cubit's approach to approximate matching relies on an accurate notion of distance between keywords. Such distance should correspond to our intuition on which keywords are similar and which are different. The choice of any particular distance is driven by domain requirements; the Cubit's core is agnostic to this choice. For example, Euclidean distance would be a reasonable choice if a keyword is a vector of network coordinates of a node (e.g. [33, 18]), whereas relative entropy would be more appropriate for representing the distance between the feature vectors of two images (e.g. [32]).

Keyword space. In this paper, we focus on keywords that are (short) text strings, such as artist names or words in a movie title. Our notion of distance targets misspellings; in particular, the distance between a given keyword and its misspelling should be small. Cubit mainly 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. We also evaluate Cubit using the *Damerau-Levenshtein distance*, an extension of the edit distance that includes the transposition of two characters (a common typo) as a single operation. Once the notion of distance is fixed – throughout the paper it will be the edit distance, unless noted otherwise – the keywords intrinsically lie in the *keyword space*, a metric space on keywords with a metric given by the edit distance.[2]

Consider a typical keyword space taken from the movie database released by Netflix [6] 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 – e.g. the median is 5, and the 90-th percentile is 9. Thus the size of a ball around a typical node grows with the radius much faster than (say) in a two-dimensional grid. In fact, a typical keyword space is very different from the "standard" metric spaces such as Euclidean space. To appreciate this difference, consider the example in Figure 1 with a set of five keywords which cannot be embedded into the coordinate plane. Such an

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

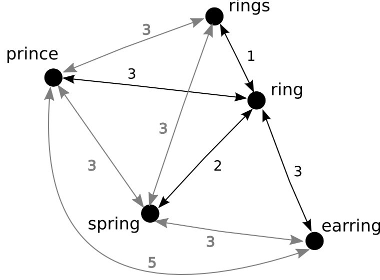


Figure 1: The edit distance between keywords: five keywords which cannot be embedded into the coordinate plane so that all distances are preserved. Preserving the distances between four nodes (all but *ring*) distorts the distances to the fifth node.

embedding becomes increasingly more inaccurate with additional keywords, even if we allow more dimensions. Cubit's use of the keyword space obviates such an inaccurate embedding.

Multi-Keyword Matching. Search queries typically consist of more than one keyword. Moreover, not all keywords relevant to the desired object may be included in the query, and some of the keywords may be misspelled. For example, a user may search for a long movie title using only a few (misspelled) keywords among those that appear in the title. Cubit matches multi-keyword queries to objects using the *phrase distance* between a query and an object, which we define as the sum, over all search terms, of the minimal edit distance between the search term and the object's keywords. (Note that the ordering of keywords does not matter.)

Cubit also supports user-specified boolean expressions over keywords and phrases to enable the construction of fine-grained, multi-keyword search queries. A phrase in the boolean expression matches an object if the phrase distance between the two is smaller than a threshold value, defined as the user-specified error probability per character times the phrase length. Objects that satisfy the boolean expression are returned to the user.

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 [26] 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.

3 FRAMEWORK

The basic Cubit routing framework builds on the small world overlay introduced in Meridian [44] for routing in the network latency space. The 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.

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 2. The

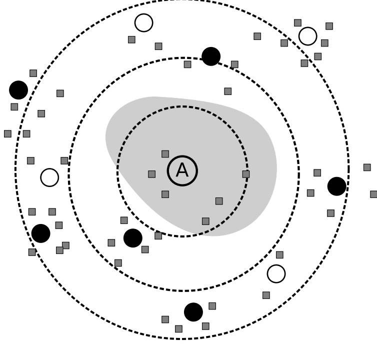


Figure 2: A Cubit node organizes its peers into concentric rings, each with a fixed number of nodes. In this example, the solid circles represent peers in node A's peer-set, the empty circles represent other nodes, and the squares represent object keywords in the system. The shaded region depicts the subspace that is closer to A than any other node. The master record for each keyword in the shaded region is stored at node A.

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 [44].

Gossip Based Node Discovery. Cubit uses a standard anti-entropy push-pull protocol [21] for node discovery and dissemination. 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. Cubit replicates objects in order to achieve 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.[3] 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. Replicas are reaped locally at the expiry of their leases.

At an even lower periodic rate, each node verifies that, for each of the secondary replicas it owns, the primary node for the replica has a copy of the object. This additional check ensures that the primary node will eventually have a copy of the object if there exists a replica somewhere in the network, limiting the impact of transient routing errors that cause incorrect initial placement of the replicas.

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 [35, 19, 38] based on object caching by intermediate nodes are not applicable to Cu-

[3] 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.

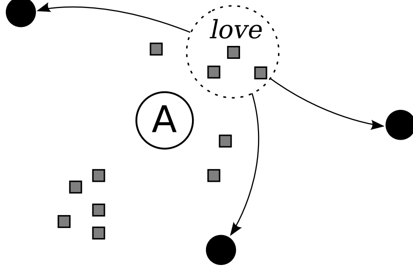


Figure 3: Cubit's load-balancing protocol prevents popular keywords from overwhelming a node. In this example, the keyword "love" is closest to node A and is generating a high degree of load. Node A creates a virtual node around the keyword, which includes its leaf set and all objects within a p edit-distance radius. This virtual node is sent to A's nearest neighbors to the keyword. Queries that arrive at these neighbors for keywords within the region can be answered without node A.

bit, as an intermediate node can not safely short-circuit a search query unless it can find an exact match. We introduce a novel 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 manufactures a virtual node at w with all the information it has on that region in the keyword space. This includes the objects in the region and the node's leaf-set, allowing the virtual node to answer queries on its behalf for that region. The virtual node is disseminated to node i's m_{off} nearest neighbors to w, which are the most likely locations to intercept and short-circuit search queries for w. Node i 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 3 illustrates the protocol.

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.

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.

Algorithm 1 SEARCH PROTOCOL

```
Require:  E: Search event             R: Local ring set
          U: Outstanding queries      H: Leaf set
 1: N ← E.GETREMOTENODE(),    I ← E.GETQUERYID()
 2: K ← E.GETFANOUT(),        T ← E.GETKEYWORD()
 3: if E.TYPE() = SearchRequest then
 4:     A ← GETKCLOSESTNODES(T, K, R + H)
 5:     N.SEND(SearchReply, I, T, A)
 6: else if E.TYPE() = SearchReply then
 7:     C ← E.GETRESULTS() - CHECKED[I] - PENDING[I]
 8:     CHECKED[I] ← CHECKED[I] + {N}
 9:     PENDING[I] ← PENDING[I] + C - {N}
10:     A ← CHECKED[I] + PENDING[I]
11:     A ← GETKCLOSESTNODES(T, K, A)
12:     if A ⊆ CHECKED[I] then
13:         for all V in A do
14:             V.SEND(FetchObjRequest, I, E.SEARCHTERMS())
15:     else
16:         for all V in A ∩ C do
17:             V.SEND(SearchRequest, I, K, D, T)
```

When a Cubit node receives an object insertion request, it concurrently issues a closest node search for each keyword using the search protocol described in the next section. The object is initially inserted at the closest node, and the the closest node further replicates the object to the f_{repl} - 1 closest neighbors to the keyword, chosen from peers in its leaf-set.

Search Protocol. For the basic (non-boolean) queries, the goal is to obtain the k_* objects nearest to the set of keywords, as measured by the phrase distance, 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 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 equidistant or 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 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.

Boolean Queries. Advanced search queries in Cubit are constructed as boolean expressions, where each term in the expression is either a keyword or a phrase. The boolean search protocol converts a boolean expression into disjunctive normal form, creating a set of conjunction clauses that are connected by OR operators. It uses the standard search primitive to find objects within a threshold phrase distance from each positive term in each clause. The standard search is modified to include all the negative terms in the same conjunction clause that act as filters, ensuring that it avoids returning objects within the threshold distance of these negative terms without requiring explicit searches on them. The union of the results from the conjunction clauses is returned.

As the number of objects matching a single keyword can be very large, the collection of intermediate results are sent as Bloom filters [15] to reduce the bandwidth requirement of the protocol. The Bloom filters are of sufficient size to distinguish between thousands of objects with a very low (0.1%) false-positive probability and requires several orders of magnitude less space than sending the actual objects. The compressed Bloom filters are usually only hundreds of bytes in size in our experiments. Since Bloom filters support union and intersection operations, all intermediate set operations can use the Bloom filters directly. The actual objects that make up the final filter are fetched from the closest nodes of the positive search terms in a final request. In this request, the closest nodes are tasked with repeating their previous search but would only return objects that are in the final filter.

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 primary replicas of objects with keywords that are closer to the new node than the node they are currently residing at. The protocol for this operates iteratively. It asks each of its k closest nodes if there are any primary replicas that should be copied to the new node that it does not already have. If at least one is closer, the protocol repeats with a larger k until no new primary replicas that should be copied are discovered.

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 can use a node ID selection protocol that deterministically constructs IDs from the IP address of the node. Each node is seeded with the same source of keywords, such as a dictionary, and the hash of the IP address 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. 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. In addition, heavier weight techniques such as PeerReview [23] can be used to identify misbehaving nodes and cleave them from the network.

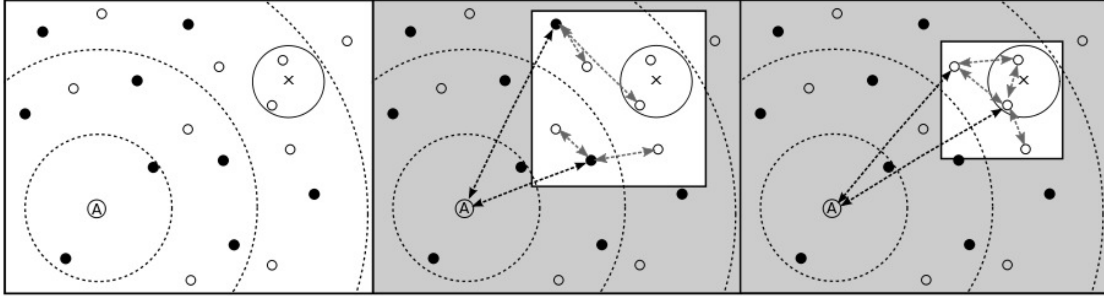


Figure 4: The Cubit search protocol iteratively zooms in on the target region. In this example, x is the location of the search term in the keyword space, the solid circles are node A's peers, empty circles are other nodes, and the circle around x includes all nodes within edit-distance q of x. Node A first finds the $n_{\min} = 2$ closest nodes to x from its peer-set, and requests their $n_{\min}$ closest nodes. Then 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.

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 provide object insert capabilities only 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 [43,20], where poorly rated objects are either discarded or are given a lower rank in response to search queries.

Sybil Attacks. Sybil attacks can allow the attackers to take control of a region of the keyword space. Countermeasures such as [30, 16] 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. The goal of this protocol is to find near-optimal matches using a small number of hops. In this section, we lay out some principled reasons why this protocol is *plausible*.[4] The state-of-the-art theoretical approach is based on *small-world networks* [26], 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.[5] A typical small-world analysis relies on the properties of the underlying graph or a metric space. The prior work 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. (These constructions discuss the *existence* of a suitable overlay, rather than a distributed construction thereof in a peer-to-peer setting.) 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.

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. As we discussed in Section 3, a typical 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. 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 the following: **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, called *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.

[4] The real-life performance is validated empirically in Section 6.

[5] See [26] for a comprehensive survey; we omit further citations.

Setup. Consider the basic *greedy 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.

We assume that the overlay is randomized. Specifically, we assume the following two properties:

(P1) node IDs are distributed *u.a.r.* [6] over Q,

(P2) 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. In Cubit, they motivate the peer-selection protocol which randomizes and diversifies the peer sets.

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

Provable guarantees. Following the literature, we would 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 let $r = d(x, q) - 1$. 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 $B(q, r) \cap B(x, r')$, for some r', if the intersection is large compared to $B(x, r')$ and contains enough node IDs; to ensure the latter, the intersection needs to be large compared to $B(q, r)$. To formalize this intuition, we define a quantity $\text{PR}(x, q)$ which measures the likelihood of making progress; we call it the *progress ratio*:

$$\text{PR}(x, q) = \max_{r'} \frac{|B(x, r') \cap B(q, r)|}{\max(|B(x, r')|, |B(q, r)|)}.$$

Using the progress ratio, we formulate a "local" guarantee for a given (x, q) pair, and then use it to prove a "global" guarantee for the search algorithm. (The proofs are omitted due to space constraints.) Both guarantees are probabilistic; we assume randomization properties (P1-P2), and the probability is over the choice of node IDs and peers. Let k_{ring} be the number of peers per ring.

[6] *u.a.r.* = *uniformly at random*. In fact, it suffices to use an approximate *u.a.r.* assumption, e.g. each element x is drawn independently with probability $p(x) \in (\frac{1}{2n}, \frac{2}{n})$, where n is the number of elements.

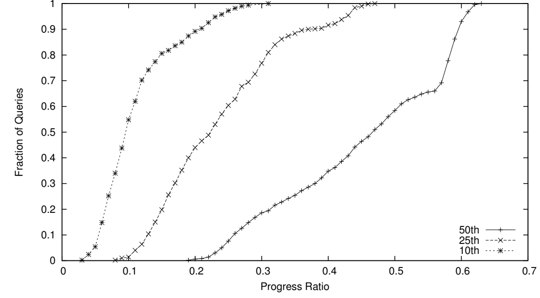


Figure 5: The progress ratios for 1000 randomly chosen node IDs and 500 randomly chosen queries. For each $p = 10, 25, 50$ we present a CDF plot for the p-th percentile progress ratio $r_p(q)$, where the CDF is taken over all queries q. For instance, a high value of $r_{10}(q)$ is a strong positive evidence: namely, for 90% of node IDs the progress ratio is *better* than $r_{10}(q)$.

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

Theorem 5.2 *Fix query $q \in Q^*$. Suppose for some $k \leq k_{ring}$ and each node x we have $\text{PR}(x, q) \geq \frac{3}{k} \log N$, 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 .*

Discussion. High progress ratio is similar to the ball growth property in [25] in that it is also a local property of balls in a metric space, and it enables a similar type of analysis. However, the property in [25] is geared towards low-dimensional grids and similar metric spaces, and is not even remotely applicable to the keyword space.

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, we considered the Netflix data set as the keyword space Q. We picked 500 queries at random from Q^*, and 1000 node IDs at random from Q, and computed $\text{PR}(x, q)$ for every id-query pair (x, q). To characterize the progress ratios relevant to a given query q, we let $r_p(q)$ denote the p-th percentile among the values $\{\text{PR}(x, q) : \text{ all nodes } x\}$, and analyze how the values $r_p(q)$ are distributed over the queries. These results are summarized as CDF plots in Figure 5.

Interestingly, the progress ratio values for the Netflix dataset are significantly higher than those for a set of random strings with the same length distribution. Naturally occurring keyword spaces differ from random ones in ways that are essential to our problem. Not surprisingly, our experiments show that the overall accuracy of Cubit on the Netflix dataset is much better.

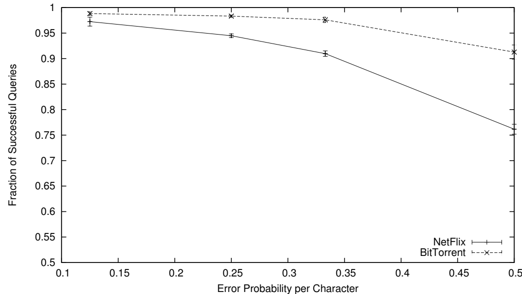
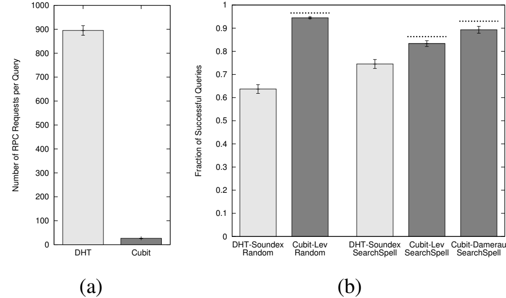


Figure 6: **Error probability per character vs. accuracy.**



(a) (b)

Figure 7: (a) **Number of RPC requests per query for a DHT-based system and Cubit.** (b) **Fraction of successful queries with keywords created from random character perturbations, and keywords created from the SearchSpell database.** The dotted lines show the metric upper-bounds.

6 EVALUATION

We implemented the full protocol described in the preceding sections as an Azureus plugin. We evaluate Cubit through both a large-scale simulation on real-world datasets and a physical deployment on PlanetLab [12].

6.1 Simulation

We use three different real-world datasets to parameterize our simulations. The first is the Netflix database [6], consisting of $17,770$ movie titles. We collected our second dataset by crawling a popular BitTorrent website for media files, consisting of over $39,000$ torrents. These two datasets represent different extremes, with the Netflix dataset providing clean input with no duplicate entries, in contrast to the much noisier BitTorrent data. Our third dataset is the CiteSeer [7] database with the titles of over $400,000$ academic papers. While not representative of file sharing content, the large dataset 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 error probability per character is the probability that, for each character in the search string, the character has been replaced by a random character. 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 higher error probability represents a more difficult query. Additionally, we evaluated search queries where keywords were modified with real human typos and misspellings from the SearchSpell database [8].

Because of the skewed distribution of English words and the conservative way we are measuring success, 100% success is not always possible. For instance, a 3-typo query for the "Lost Ark" includes "Last Orc" and might legitimately return an entirely different set of objects; it is possible for the intended object to not be present among the search results if the randomly introduced typos veer into a different portion of the keyword space. We nevertheless retain this conservative success criteria, and indicate the highest achievable success rate as the *metric upper-bound* where applicable.

In the following experiments, unless specified otherwise, each test consists of 4 runs of 1024 nodes, 10 nodes per ring (88.2 peers per node on average), an error probability per character of 0.25, 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.

Accuracy. 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. Our accuracy metric is equivalent to recall, a common statistical classification used in information retrieval [7]. Figure 6 shows that Cubit can successfully answer queries where a third of the characters in the search string is expected to be erroneous with more than 90% accuracy. Surprisingly, for queries where half the characters in each search keyword are expected to be perturbed, Cubit is still able to successfully resolve them more than 75% and 90% of the time for the Netflix and BitTorrent datasets respectively. Cubit achieves a higher accuracy on the BitTorrent dataset because the average number of words of a BitTorrent title is 6.6, nearly twice that of a Netflix title at 3.6. A higher number of words per title provides proportionally more

[7] One cannot meaningfully present precision, the complementary classification to recall, as our datasets do not include relevance information between objects.

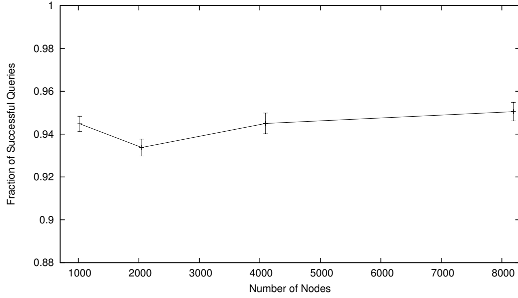


Figure 8: **Number of nodes vs. accuracy.**

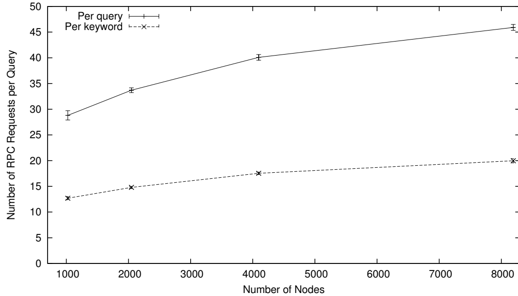


Figure 9: **Number of nodes vs. RPC requests per query.**

keywords per query which improves search accuracy.

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 7(a). We use a basic DHT implementation based on Pastry [39] 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 [46], enables approximate matching without resorting to searching for every possible spelling permutation. Figure 7(b) shows that this approach achieves a success rate of 64% for keywords with random character perturbations and 75% for keywords taken from the SearchSpell database where some of the misspellings are phonetically similar. Cubit using the standard Levenshtein distance achieves over 94% accuracy for random character perturbations and 83% accuracy for SearchSpell misspellings. The drop in accuracy is because phonetic misspellings and character transpositions which make up a significant portion of the SearchSpell database

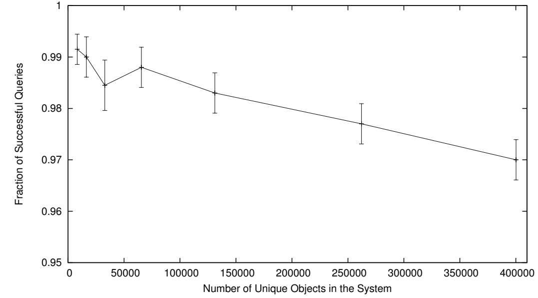


Figure 10: **Number of objects in the system vs. fraction of successful queries using the CiteSeer dataset.**

do not match well with Levenshtein distance. Even an ideal, centralized service struggles to meet our definition of accuracy when using the Levenshtein distance for matching, with a metric upper-bound accuracy of 97% and 86% for keywords with random character perturbations and SearchSpell misspellings respectively.

In contrast, Cubit using the Damerau-Levenshtein distance, which counts the transposition of two characters in a keyword as a single operation, reduces the distance between the misspellings and their origin word and improves Cubit's accuracy to just under 90% for SearchSpell misspellings with a metric upper-bound of 93%. Damerau-Levenshtein based Cubit performs identically with the same overhead in all other experiments as the Levenshtein version; the transposition operation only minimally affects the edit-distance between correctly spelled words, and randomly generated misspellings and their origin word. These results illustrate the importance of pairing a distance metric that closely matches the expected type of errors.

Scalability. 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 8 shows that search accuracy is largely independent of the system size, with Cubit maintaining a 94% accuracy across the tested system sizes. These results indicate that the greedy search protocol very rarely terminates prematurely as a small increase in the hop length due to an increase in system size has a negligible effect on the accuracy. Figure 9 shows how the number of RPC requests per query and per searched keyword grows with the number of nodes. The growth rate is proportional to the maximal number of hops times the number of nodes per ring. The former is upper-bounded by a small constant, while the latter is set to $\log(\#\text{nodes})$ yielding logarithmic scaling.

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 dataset consisting of more than $400,000$ academic

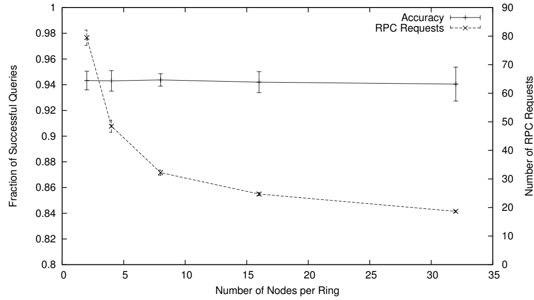


Figure 11: **Number of nodes per ring vs. accuracy and the number of RPC requests per query.**

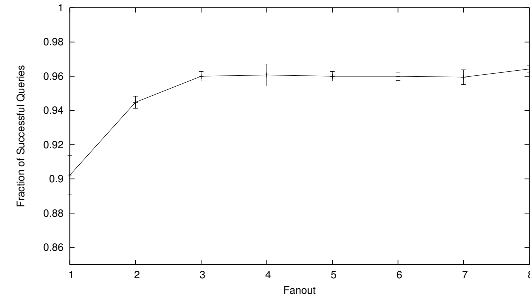


Figure 12: **Search fanout vs. accuracy.**

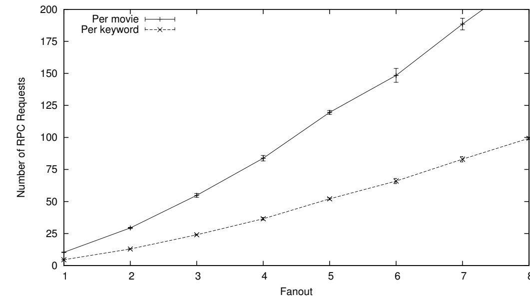


Figure 13: **Search fanout vs. the number of RPC requests.**

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 10 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 dataset is considerably higher than on the Netflix dataset. This is primarily due to the relatively longer, more distinctive titles found in academic papers, resulting in a sparser, more search friendly keyword space.

A significant concern with boolean search queries is the large number of intermediate search objects that must be sent between nodes. Cubit uses Bloom filters as intermediate storage in order to reduce the bandwidth overhead. To verify the technique's effectiveness, we examine the bandwidth requirement of boolean searches on the Netflix dataset which on average contains 3.6 keywords per object. In our experiments, we use the conjunction of two-thirds the total keywords of the object as the boolean search expression. The median bandwidth used per boolean search query is only 17 KB with only trace amounts of false positives.

Parameters. 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 and query performance. A low nodes per ring value provides poor coverage of the space and requires more RPC requests to complete a query, where a high nodes per ring value requires additional state to be kept and maintained at each node. Figure 11 shows that accuracy is mostly unaffected by the number of nodes per ring, in contrast to the number of RPC per query which decreases dramatically from two nodes per ring to four, and flattening out at sixteen nodes per ring.

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 12 illustrate that increasing fanout from one to two nets a 4% improvement in accuracy, with further increases netting subsequently smaller gains. However, the accuracy comes at the cost of requiring additional RPC requests. Figure 13 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 14 shows that increasing replication from one to four increases search accuracy by more than 12%. Increasing replication beyond four gives only marginal accuracy improvements.

Replication and churn. The object replication factor trades off accuracy against bandwidth for replica management. Figure 14 shows that increasing replication from one to four increases search accuracy by more than 12%. Increasing replication beyond four gives only marginal accuracy improvements. The 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 1.6 KB/s for each Cubit node, which compares favorably to 2 KB/s used for the maintenance of the BitTorrent DHT on a host

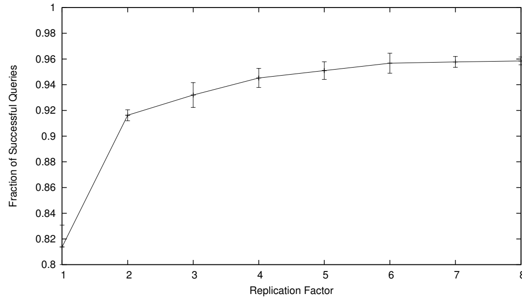


Figure 14: **Replication vs. accuracy.**

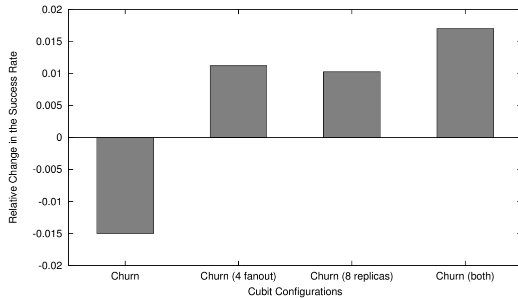


Figure 15: **Relative change in accuracy due to churn.** Realistic churn rates have modest effects. Increased fanout or object replication (or both) can compensate for the effects of churn.

with a 256 Kb/s upstream connection [9].

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 query fan-out and replication factor, provides ways to maintain search accuracy under higher levels of churn.

Load-balancing. 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 random 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 16 shows that the Cubit load-balancing protocol is effective at reducing the load at request hotspot through

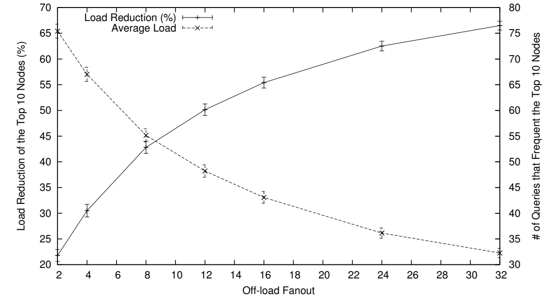


Figure 16: **Offload fanout versus load at hotspots.**

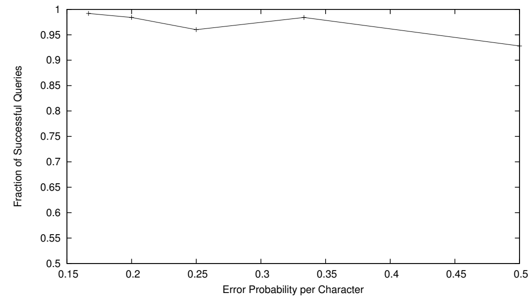


Figure 17: **Error probability per character versus the accuracy in the Azureus deployment.**

the introduction of virtual nodes. Even an off-load fanout of 8 can 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 trackerless torrents 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 [37].

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 with different error probability per character values. Figure 17 shows that Cubit can successfully answer queries where half the characters are expected to be perturbed with more than 90% accuracy which closely matches our simulation predictions. There are currently more than $6,000$ active users. The plugin is available at our project website at http://www.cs.cornell.edu/~bwong/Cubit.

7 RELATED WORK

Cubit is a system for performing approximate matching in peer-to-peer overlays. We separate past work into two categories, routing in overlay networks and approximate matching techniques, and survey both in turn.

Routing in overlay networks. Cubit is a loosely structured overlay network that most closely resemble a distributed hash table. It differs from previous DHTs [39, 41,47,36,29,24] by providing a novel approximate match primitive rather than supporting only precise lookups.

Query routing in Cubit is similar to routing in CAN [36], SWAM [11], and Meridian [44]. 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-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 [11] 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 for the node join protocol.

While the Cubit framework builds on top of Meridian [44], a system for resolving network-latency related queries, the two systems differ inherently and significantly in the problems they address, the way they perform routing, and the kinds of optimizations they employ. The Cubit framework is much more general in that it supports distance metrics beyond network latencies, and enables significantly more complex boolean search queries. Since Cubit queries are more complex because they necessitate finding the set of *all* nodes that meet a particular constraint, and because Cubit nodes constitute a key/value database instead of the more constrained node/latency space, the protocols for node join and query routing are significantly different. Cubit also introduces optimizations not applicable to the Meridian context, such as mechanisms to proactively maintain object replication for improved resiliency in a highly dynamic peer-to-peer environment, and to encapsulate and offload keyword regions to nearby neighbors.

Several peer-to-peer systems, e.g. [41, 28, 27], use overlay routing based on the small world networks [26]. These systems use a specific virtual space (e.g. a ring) in which long links are introduced such that a simple greedy routing protocol can find short routes. These systems are inherently limited to precise lookups. A related line of work considers small-world networks on arbitrary underlying spaces, see [26] 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.

Approximate matching. An alternative design (proposed in [45]) involves representing keywords as points in a low-dimensional Euclidean space. Once nodes and keywords are embedded, techniques such as CAN [36] and Meridian [44] can be used for navigation in that space. While this approach gives a clean and intuitively appealing representation of the keyword space, the literature on metric embeddings does *not* provide embeddings of edit distances into Euclidean space – even with high dimension – that are known to have a sufficiently high precision for the approximate matching of short keywords. The embedding is prohibitively inaccurate in practice, distorting the navigation.

Past work has proposed to use the Soundex algorithm to encode keywords by their phonemes before indexing them in a DHT [46]. Unlike edit distance, Soundex is appropriate only for English keywords and is not effective against typing errors. Our evaluation compares the performance of Cubit versus Soundex; while a Soundex lookup is simpler, Cubit is more accurate.

DPMS [10] 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 [15]. 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 [40] 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, 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 [31], 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 [13], which uses locality-sensitive hashing to cluster similar terms. The system is primarily focused on finding similar documents rather than keywords.

pSearch [42] uses latent semantic indexing on documents to generate vectors that represent its relative similarity to other documents in the system. CAN [36] 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 set of search terms.

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 datasets. 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 significantly more accurate than systems based on Soundex hashing. 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. This overall approach may be applicable to other distributed domains where a similarity-based clustering of objects is desired.

References

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

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

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

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

[5] The Pirate Bay Trial: The Official Verdict - Guilty. http://torrentfreak.com/the-pirate-bay-trial-the-verdict-090417.

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

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

[8] Searchspell. http://www.searchspell.com/typo/.

[9] Advanced Bittorent. http://www.bittorrent.com/btusers/guides/-bittorrent-user-manual/appendix-bittorrent-mainline-interface/preferences/advanced#dht.rate.

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

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

[12] 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. In *NSDI*, 2004.

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

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

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

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

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

[18] F. Dabek, R. Cox, F. Kaashoek, and R. Morris. Vivaldi: A Decentralized Network Coordinate System. In *SIGCOMM*, 2004.

[19] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica. Wide-Area Cooperative Storage with CFS. In *SOSP*, 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. In *CCS*, 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. In *PODC*, 1987.

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

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

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

[25] D. Karger and M. Ruhl. Finding Nearest Neighbors in Growth-Restricted Metrics. In *STOC*, pages 63-66, 2002.

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

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

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

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

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

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

[32] H. Neemuchwala and A. Hero. Image Registration in High-Dimensional Feature Space. In *Computational Imaging*, 2005.

[33] T. Ng and H. Zhang. Predicting Internet Network Distance with Coordinates-Based Approaches. In *INFOCOM*, 2002.

[34] A. Parker. P2P in 2005. 2006.

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

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

[37] 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. In *SIGCOMM*, 2005.

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

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

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

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

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

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

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

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

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

[47] 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, 2001.