Efficient Peer-To-Peer Lookup Based on a Distributed Trie

Michael J. Freedman[1] and Radek Vingralek[2]*

[1] MIT Lab for Computer Science, 200 Technology Sq., Cambridge, MA 02139 USA
email: mfreed@lcs.mit.edu
[2] Oracle Corp., 500 Oracle Parkway, Redwood Shores, CA 94065 USA
email: radek.vingralek@oracle.com

Abstract. Two main approaches have been taken for distributed key-value lookup operations in peer-to-peer systems: broadcast searches [1, 2] and location-deterministic algorithms [5–7, 9]. We describe a third alternative based on a distributed trie. This algorithm functions well in a very dynamic, hostile environment, offering security benefits over prior proposals. Our approach takes advantage of working-set temporal locality and global key/value distribution skews due to content popularity. Peers gradually learn system state during lookups, receiving the sought values and/or internal information used by the trie. The distributed trie converges to an accurate network map over time. We describe several modes of information piggybacking, and conservative and liberal variants of the basic algorithm for adversarial settings. Simulations show efficient lookups and low failure rates.

1 Introduction

We describe a set of algorithms for key-based lookup in a distributed system consisting of a number of uniform *peers*. A lookup service is a necessary component for peer-to-peer file-sharing systems, which need to map filenames to the location of peers that store them. Such an algorithm may return the files themselves or the addresses of servers storing of files.

Most lookup algorithms deploy a *lookup structure* to efficiently locate values associated with keys. The lookup structure can be organized as an (extendible) hash table, a trie, a binary tree, a B-tree, etc. A distributed lookup algorithm trades off the efficiency of lookups for the maintenance of the lookup structure. Maintenance is initiated by either a key/value pair insert or by a membership change.

Lookups can be made very efficient by replicating the lookup structure on every peer. However, maintenance is slow as all peers must keep their lookup structure replicas consistent. To date, peer-to-peer systems have taken two approaches to reducing the maintenance costs.

* This work was partially done while both authors were employed by InterTrust Technologies, STAR Lab, 4750 Patrick Henry Drive, Santa Clara, CA 95054 USA.

The first approach, adopted by Gnutella [2] and Freenet [1], eliminates the lookup structure altogether and thus requires no maintenance. However, lookups are implemented by a broadcast-like search on all known peers, costing efficiency and scalability. (Freenet aims to reduce overhead by broadcasting first to neighbors that previously returned similar keys and replicating key/value pairs along the entire path between sender and receiver.)

The second approach partitions the lookup structure and distributes a subset of partitions on each peer. Since maintenance updates are typically localized, peers update only a smaller number of partition replicas. The system assigns partitions to peers either statically or dynamically. In the static-assignment approach, peer addresses are mapped to the key space and each node replicates only those partitions that are "close" to its address. Systems such as Chord [7] (mapping via consistent hashing), CAN [5] (routing on a d-torus), and Tapestry [9] and Pastry [6] (both related to Plaxton trees [4]) adopt this model. The static assignment enables efficient message routing, with the worst case message overhead limited by a logarithm of the network size (assuming no updates to the lookup structure). However, if peers join and leave the network rapidly, the lookup structure may never stabilize to a state that enjoys this provably logarithmic bound. Furthermore, the routing overhead does not adapt to the workload. Consequently, peers will always incur a higher message overhead to access "far away" parts of the key space. Moreover, the peers must dedicate resources for upkeep of the routing information that they do not use. Finally, it is easy for an attacker to target a specific part of the key space because she can determine (using the static mapping) the peers that hold the corresponding partitions of the lookup structure.

When the partitions are assigned dynamically, the distribution of the partitions can adapt to the workload by having peers replicate only the partitions that they frequently access. It is also harder to attack a particular part of the key space because it may be impossible to determine locations of all replicas of a particular partition without inspecting routing tables of all peers. However, the same property makes efficient routing more difficult.

Relaxing the consistency criteria for partition replicas can further reduce maintenance costs. However, it can happen that peers hold stale replicas if they update local lookup structures lazily [3, 8]. They can commit addressing errors when requesting values from peers that are unavailable or that no longer hold the values. To ensure that the extra cost of addressing errors does not approach the cost of a broadcast, peers must limit addressing errors by piggybacking the updates on other traffic. Then, peers reconcile conflicting updates to achieve replica convergence.

We present a set of algorithms that exploit both *dynamic partitioning* based on peers' access locality and *lazy updates* of the lookup structure to reduce maintenance cost. All algorithms piggyback trie state on lookup responses. Peers use timestamping to reconcile conflicting updates. The algorithms differ in the volume of the trie structure piggybacked and how aggressively the requester uses these partitions.

These algorithms do not preclude *strong anonymity* for either peers initiating a lookup or peers storing the corresponding value. The trie structure can index endpoints of fixed-length mix-net circuits, as opposed to the desired peers. The mix-net circuit will relay messages from an endpoint to an anonymized recipient. Therefore, our lookup algorithms (as well as Chord, CAN, Tapestry and Pastry) can provide anonymity at a constant cost of extra messages, by treating anonymity as a goal for the underlying communication channel.

2 System model

The system consists of n peers that all implement the following interface:

- `lookup( key )`. The callee sends to the caller the value associated with `key` if successful or a failure message. In both cases, the callee may piggyback additional state on the response (algorithm dependent).
- `insert( key, value )`. The callee inserts a $<$ `key,value` $>$ pair into its lookup structure.
- `join()`. The callee sends to the caller initial state needed to bootstrap `lookup` operations.

Peers cannot update or delete values that were previously inserted, although they can re-insert new versions under a different (or even the same) key. The caller is responsible for ensuring key uniqueness, if required. Peers join the system only intermittently, where "join" is defined as an *initial* entry into the network, not a reconnection after failure. Peers can leave at any time or fail silently; no maintenance operations are necessary in either case.

Each peer stores a number of key/value pairs locally, that were either inserted or looked up by the peer. The peer also stores partitions of a lookup structure organized as a trie, as shown in Figure 1. A trie representation is insensitive to the insertion ordering. Consequently, it is easier to merge two incomparable versions of the lookup structure.

Internal trie nodes consist of 2^m *routing tables.* Each routing table consists of l *entries.* An entry consists of a peer address a and a timestamp t. Each level of trie node "consumes" m bits of the k-bit key. If the node is a leaf (defined as having depth $\lceil k/m \rceil$), then the entry in its matching routing table indicates that peer a was known at time t to hold the specified value. Otherwise, the entry at its ith routing table indicates that peer a was known at time t to hold a replica of the ith child of the node. The timestamps are generated locally by each peer; we assume loose clock synchronization (say, to within a few hours).

Each peer stores only the subset of trie nodes corresponding to its access pattern. However, all peers maintain the following invariant:

> *Ancestor invariant*: If a peer holds a trie node, it must hold all ancestors of the node.

We represent trie paths more compactly by explicitly relying on this invariant: If a peer appears in some routing table, it is known to hold not only the

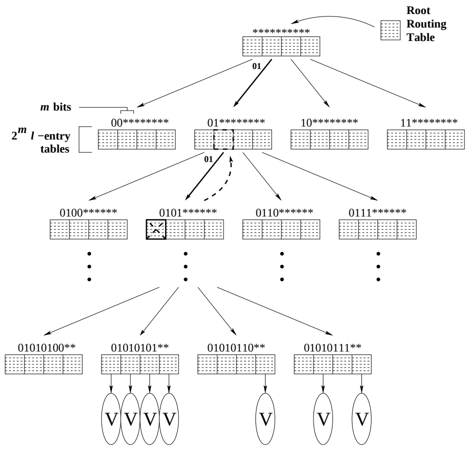

Fig. 1. Trie lookup structure ($k = 10, m = 2$).

node's child, but also an entire path down to that child. Peers naturally maintain this invariant by replacing trie nodes and routing table entries based on access frequency. Logically, nodes closer to the trie root are more widely replicated by peers, removing any single point of failure.

3 Algorithms

The algorithms we present for a distributed trie lookup share most of the basic steps. They differ only in what state is piggybacked on lookup responses and how this state is used. We first describe the basic framework shared by all algorithms and subsequently the differences.

In order to join the distributed system, a peer needs to know the address of at least one participating peer, called its *introducer*. The introducer responds to a join request with its own root routing table, as labeled in Figure 1. The new peer uses this root routing table as a bootstrap mechanism when it exhausts the entries within its own trie during a lookup.

Insertion is performed locally by inserting the key/value pair (and the corresponding path from the root of the trie) into local storage. Alternatively, the peer could send an insert request to other peers, similar to [1]. The peers may decline to store the inserted pair. This paper does not consider such an insert operation, in order to focus solely on the effect of piggybacking state in lookups.

The lookup caller first checks local storage for the value corresponding to the lookup key. If present, the lookup process terminates. Otherwise, the caller initiates a distributed lookup process. We present an example of `lookup(0101000000)` for comprehension.

Caller A searches its local trie for a routing table that most specifically matches lookup key. Such a routing table `010100*` is shown with a solid box in Figure 1. Subsequently, caller A then sends a lookup query to peer B, who has the latest timestamp in the routing table. Peer B is most recently known (to Peer A) to hold the child trie node `010100*` (that A does not currently have). B returns the actual value if B holds it. B's response may either contain additional trie state or be a failure with no state. If B returns a deeper routing table, A drops down to that level and repeats this process. If B returns failure, caller A tries other peers in its routing table in decreasing timestamp order. If all such peers fail, A backtracks in its local trie and repeats the same process on the parent routing table. The figure shows a dashed line: routing table `010100*` failed and is crossed out, and A backtracked to routing table `0101*`.

Recall that each trie node's routing tables are of maximum size l. Once the caller starts backtracking, we enumerate larger "virtual" routing tables: an entry list containing all peers thought to hold the desired child trie node. Peer A's virtual view of `0101*` is that level's actual routing table merged with all tables in its subtrie (minus the entries in table `010100*` already contacted). By the ancestor invariant, the peers holding trie nodes in the subtrie must also have a copy of the higher-level routing table. Therefore, we effectively increase the size of the higher-level routing tables without additional storage. If an entry's routing table is full when new entries are installed, the least recent entry is evicted from this table and propagates up the trie according to timestamp.

Peers may backtrack their local tries during lookup up to the root routing table. If the value is not found at this time, the lookup process terminates with a failure.

3.1 Bounded, unbounded, and full path modes

The *bounded*, *unbounded*, and *full path* modes explore the tradeoff between the size of piggybacked trie state with the speed of convergence of peers' tries to an accurate network map.

In *bounded mode*, the callee responds to a lookup with its most specific routing table matching the key (or the value itself), provided its routing table is more specific (deeper) than the caller's current table. Otherwise, the callee responds with a failure. The caller integrates the more specific routing table into its local trie and proceeds with the lookup on this table.

In *unbounded mode*, the callee responds with its most specific routing table for the key, regardless of how this compares to the caller's current routing table. This additional state is useful to pre-fetch information about new peers or more recent locations of higher-level trie nodes to be used when backtracking. The caller integrates the returned routing table into its local table at that same depth, by selecting the l most recent distinct entries from the two tables.

In *full path mode*, the callee responds with the entire path (consisting of routing tables) from the root table to its most specific routing table for the key. The caller integrates the path into its local trie using the same mechanism as in the unbounded mode.

3.2 Conservative and liberal modes

Most peer-to-peer lookup algorithms are susceptible to malicious behavior. We describe one particular attack conceptually similar to DNS cache poisoning and propose a conservative mode to resist this attack.

A malicious peer can effectively suppress access to a value by falsely advertising the availability of some key (with a recent timestamp) and then dropping lookup requests. A set of l malicious peers can cause innocent peers to completely replace the entries in their routing tables with malicious peer addresses. While backtracking can help route around this problem by finding less-specific routing tables, these malicious peers have caused the system to lose efficiency.

We propose a verified-only update heuristic for a *conservative mode*. Namely, callers update their local trie with only the new entries that transitively led them to the desired value. This assumes that peers can verify the validity of returned data. For example, if peer B returns {C,D,E} to peer A at depth 1, peer C returns {F,G,H} to A at depth 2, and peer F returns the actual value, peer A updates his trie only with peers B, C, and F in the corresponding-depth routing tables. Conservative mode ensures that entries in our routing tables have performed useful work in the past. Therefore, we hope they will continue to be useful.

In *liberal mode*, callers immediately update their local tries with any piggybacked state.

4 Preliminary experimental evaluation

We implemented a simulator to compare the performance of algorithms modes described in Section 3. We present *preliminary* simulation results in this section.

We simulated a system consisting of 200 peers. The tries maintained by the peers were characterized by parameters $k = 10$, $m = 2$ and $l = 10$. Each experiment started with an initial loading phase, when the peers inserted a total of 2,000 randomly-generated key/value pairs in the system. These pairs were distributed randomly among all peers. The keys were uniformly distributed in $[0, 2^k - 1]$. Subsequently, at each simulated time-step, we issued a lookup to a randomly-selected peer and dynamically changed the membership by removing peers or adding new ones with a probability 0.005.

For each lookup phase we collected the following statistics:

- *Message overhead.* We classify lookups as either local (*i.e.*, those that could be satisfied by a peer locally) or remote (*i.e.*, those that required sending lookup operations to other peers). For each remote lookup, we measured the number of lookup operations that were sent to other peers (transitively) in order to satisfy the request.
- *Failure probability.* For each lookup, we measured the probability of its failure. A lookup fails when the requesting peer's trie did not contain sufficient information to locate an existing key/value pair (even after contacting other peers).

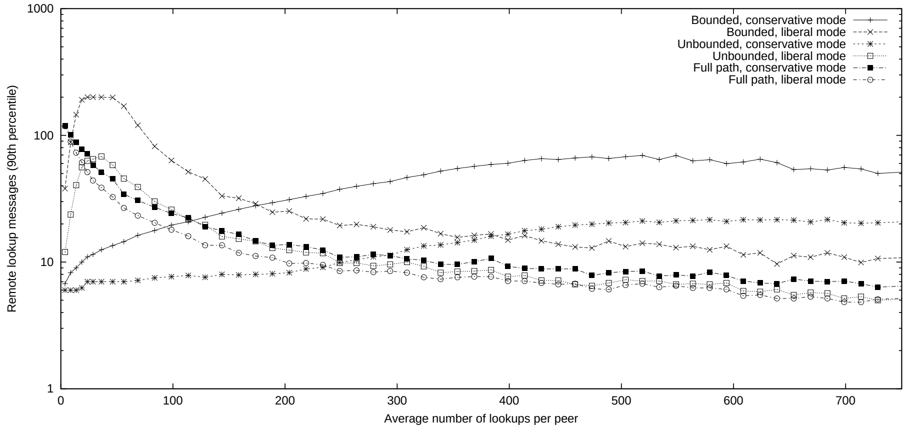

Fig. 2. The 90^{th} percentile of the number of messages generated by a remote lookup.

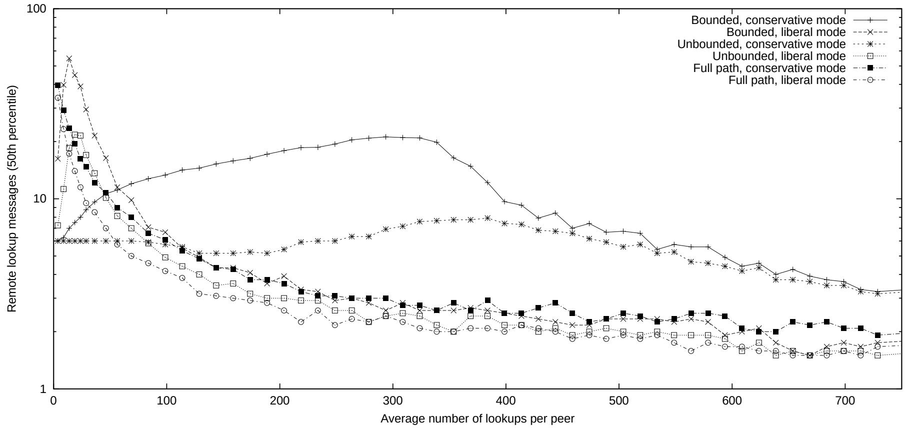

Fig. 3. The 50^{th} percentile of the number of messages generated by a remote lookup.

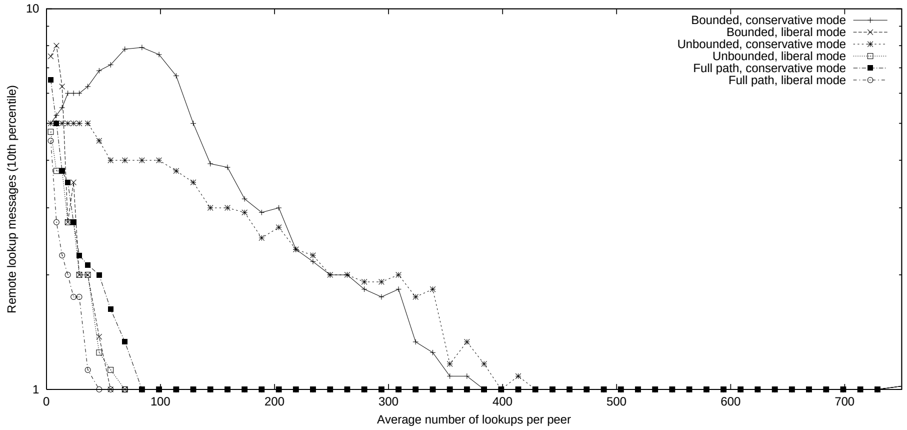

Fig. 4. The 10^{th} percentile of the number of messages generated by a remote lookup.

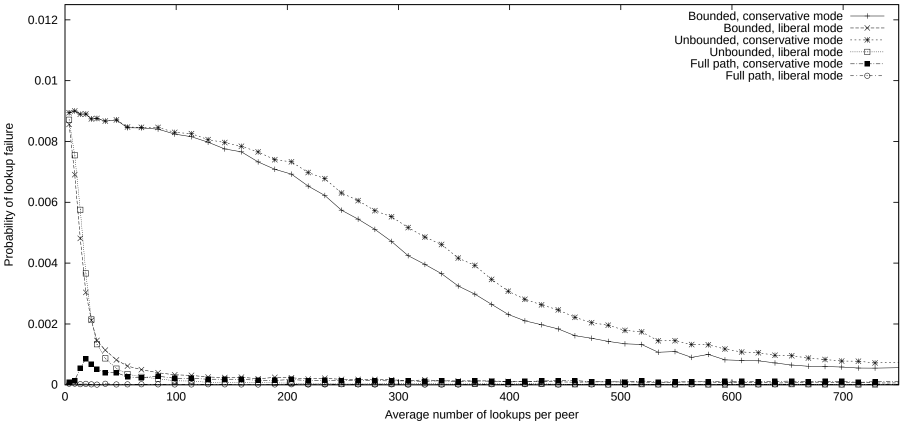

Fig. 5. The probability of lookup failure.

The message overheads of all algorithm modes are found in Figures 2, 3, and 4. The three figures respectively show the 90^{th}, 50^{th}, and 10^{th} percentiles of the number of query/response message pairs (*i.e.*, lookup operations) that were generated by a remote lookup. The reported values are conservative, as the message overhead explicitly excludes lookups that were satisfied locally. We also did not allow peers to benefit from access locality by generating keys according to a skewed distribution. The failure probabilities of all algorithm modes are found in Figure 5.

Based on the limited test data, we make the following conjectures:

- The message overheads of all modes converge toward less than $\log n$ lookup operations per remote lookup. The failure probability of all modes converge toward zero. Thus peers seem capable of correctly learning a recent view of the distributed trie.
- There is an implicit trade-off between message size and convergence time to low message overhead. Full path mode converges the fastest, yet sends the largest messages. Bounded mode converges the slowest, but with the benefit of sending smaller messages.
- The bounded and unbounded conservative modes converge slower than their liberal counterparts. (The increases in message overhead for the conservative modes, as shown in Figure 2, is a consequence of the learning process: While a lookup might fail after a node contacts its few known peers at the beginning, it will exhaustively contact its increasing number of known peers as the experiment progresses.) Consequently, there is a performance cost for reducing risk of a system infiltration by malicious peers.
- The full path conservative and liberal modes converge with almost the same rates. Therefore, if peers exchange enough state, they can reduce the risk of malicious system infiltration with only small performance loss.

In addition to these experiments with dynamic membership, we ran a similar series of experiments with static membership. The results were practically the same as those reported in the included figures (we exclude the corresponding graphs due to space constraints). Therefore, we conjecture that all of the algorithm modes are relatively resilient to system membership changes.

5 Conclusions

We propose a new approach to key/value lookup in peer-to-peer systems based on a distributed trie. Compared to broadcast-based routing, the algorithms clearly lead to a lower message overhead.

Compared to the static-partitioning-based (or location-deterministic) approaches, these algorithms can deliver a lower message overhead, given sufficient time for peers to learn the distribution of frequently-accessed keys. Message overhead does not depend on the proximity between the peer address and the looked-up key, nor are peers required to maintain state about regions of the keyspace that they do not access. Our algorithms do not assume a globally known mapping of peers address space into the key space. Consequently, it is more difficult for an adversary to target all replicas of a particular key/value pair because it is impossible to determine all locations of a particular key/value pair without inspecting routing tables of all peers.

On the other hand, our algorithms can degenerate to a broadcast for peers with very stale views, while the static-partitioning-based approaches have logarithmic upper bounds on message overhead per lookup. (Although the logarithmic upper bounds assume absence of updates to the lookup structure.)

We believe that both static and dynamic partitioning algorithms have their merits. The static algorithms provide a better worst case behavior and typically require peers to store a smaller state. The dynamic algorithms can adapt to the workload and are more resilient to certain types of attacks. We conceive that both kinds of algorithms may be used in conjunction: A peer may send a limited number of lookup operations based on its local trie and, only if all failed, revert to a static-partitioning-based lookup. Symmetrically, if the static-partitioning algorithms seek to reduce their (fixed) message overhead by caching additional state, they can use our algorithms to maintain consistency of the cached state.

References

1. Ian Clarke, Oscar Sandberg, Brandon Wiley, and Theodore Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proceedings of the Workshop on Design Issues in Anonymity and Unobservability*, pages 46–66, July 2000.
2. Gnutella website. http://gnutella.wego.com.
3. W. Litwin, M. Neimat, and D. Schneider. *lh** - linear hashing for distributed files. In *In Proceedings of the ACM SIGMOD Conference*, May 1993. Washington, DC.

4. C. Plaxton, R. Rajaraman, and A. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proceedings of the ACM SPAA*, pages 311–320, June 1997.
5. S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. ACM SIGCOMM*, San Diego, 2001.
6. Antony Rowstron and Peter Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proceedings of the 18th IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001)*, November 2001.
7. Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. ACM SIGCOMM*, San Diego, 2001.
8. R. Vingralek, Y. Breitbart, and G. Weikum. Distributed file organization with scalable cost/performance. In *In Proceedings of the ACM-SIGMOD International Conference on Management of Data*, May 1994. Minneapolis, MN.
9. Ben Zhao, John Kubiatowicz, and Anthony Joseph. Tapestry: An infrastructure for fault-tolerant wide-area location and routing. Technical Report UCB/CSD-01-1141, Computer Science Division, U. C. Berkeley, April 2001.