LOT: A Robust Overlay for Distributed Range Query Processing

André Allavena Qiang Wang Ihab Ilyas Srinivasan Keshav
University of Waterloo
{aallavena, q6wang, ilyas, keshav}@uwaterloo.ca

Technical Report CS-2006-21

Abstract

Large-scale data-centric services are often handled by clusters of computers that include hundreds of thousands of computing nodes. However, traditional distributed query processing techniques fail to handle the large-scale distribution, peer-to-peer communication and frequent disconnection. In this paper, we introduce LOT, a robust, fault-tolerant and highly distributed overlay network for large-scale peer-to-peer query processing. LOT is based on a robust tree overlay for distributed systems. It uses virtualization, replication, geographic-based clustering and flexible state definition as basic design principles. We show how we map these principles to desirable performance goals. Moreover, we provide a light-weight maintenance mechanism for updating state information. Analysis and simulations show that our approach is superior to other well-known alternatives in its query processing performance and handling of churn.

1 Introduction

Large-scale data-centric applications are often handled by clusters of computers grouped in data centers that include hundreds and thousands of computing nodes. Large organizations usually have several data centers distributed around the globe. Although there are many different deployments, we assume the following generic distributed environment: (1) the computing nodes are commodity hardware that are similar in their storage and processing capabilities; (2) nodes communicate with each other in peer-to-peer mode for query processing through an underlying network; (3) communication among nodes within the same data center is much cheaper than across data centers; and (4) failures (*e.g.*, disconnection or power shortage) cause nodes to become up and down at arbitrary times.

These properties pose several query processing challenges, where traditional distributed query processing techniques fail to handle the large-scale distributions, the peer-to-peer communication and the frequent disconnection. Several peer-to-peer data layout and query processing algorithms have been introduced in recent years [16, 14, 8, 7, 6, 3]. However, these attempts usually focus on one aspect of the problem and fail to take into account other challenges. For example, while DHT-based approaches [16, 14] focus on load balancing, they fail to provide efficient support of range queries. Tree-based structures [8, 7, 6, 3] provide a scalable indexing mechanism with efficient support of range queries, however, these techniques usually suffer from the intrinsic weaknesses of tree structures with respect to robustness and fault tolerance.

In this paper, we introduce LOT, a robust and highly distributed overlay for efficient and reliable query processing.

1.1 Design Principles

LOT is a virtual B-tree-like structure, which guarantees routing paths and query processing delays to be logarithmic in the network size. In designing LOT, we exploit four principles that map to our performance goals:

- *Geographic-based clustering* Servers are increasingly being placed in well-managed, highly-connected data centers, where each data center contains tens or hundreds of thousands of servers. From the perspective of a server, every other server within the same data center is essentially at zero distance, in terms of latency and bandwidth, compared to a server at another data center. We believe that algorithms should recognize and exploit this link heterogeneity. From an algorithmic perspective, this means that it is acceptable to greatly increase 'local' traffic (*i.e.* within a data center) in order to decrease

communications between data centers. LOT exploits this geographic proximity in deciding on node joining strategies, and by using more intra-cluster communications in completing expensive operations such as range index updates and structural changes.

- *Virtualization* Load-balancing is an important challenge for tree-structured overlay network since upper level nodes tend to get more workload and become a bottleneck. LOT solves this problem by arranging all nodes at the leaf level and by using them to emulate upper-level virtual nodes. By 'emulation', we mean that a computing node acts on behalf of all of its ancestor virtual nodes in query processing and overlay network maintenance tasks. Therefore, on average, all servers emulate the same number of upper-level nodes and share an even workload.

- *Replication and multi-path routing* An important aspect of massively parallel computing is that each server is less reliable than in a traditional mainframe approach. Therefore, algorithms should recognize that a non-trivial fraction of servers may be down at any point in time. LOT uses replication and virtualization, and exploits multiple paths in the routing structure, to increase system availability. We prove that the system fails with a very low probability bounded by $\mathcal{O}((\frac{1}{N})^{c\frac{(1+\rho)}{2}})$, where c is a constant and ρ is a robustness parameter at most poly-logarithmic to the network size.

- *Flexible state definition* The LOT virtual tree structure has a flexible state definition that allows it to be used in a variety of applications, *e.g.*, in-network function computation, and distributed indexing mechanisms to support the computation of range-queries, top-k queries and other data-centric applications.

1.2 Contribution

To the best of our knowledge, we are the first to analyze the emerging data center infrastructure and develop a distributed and fault-tolerant index based on a tree overlay for efficient range query processing over this infrastructure. LOT is distinguished for its strong guarantee on efficiency, load-balancing and robustness. We summarize the main contributions of our work as follows:

- We introduce the structure of LOT and we show how to achieve robustness, load-balancing through a non-trivial use of node virtualization, proximity-based clustering and multi-path routing.

- We show how to efficiently use LOT to answer range-queries as an example of query processing challenges for highly-distributed peer-to-peer environments.

- We introduce a simple yet efficient maintenance technique that achieves resilience to node failures and arbitrary join/leave patterns of data nodes.

- Through extensive experiments, we demonstrate the effectiveness of our design and show its superiority over previous work.

The organization of the paper is as follows. We introduce the LOT framework in Section 2. In Section 3 we describe how to use LOT to support range queries. We present our performance evaluation results in Section 4. Other applications of LOT are described in Section 5. We discuss related work in Section 6 and conclude in Section 7.

2 LOT Framework

For clarity of presentation, we use "p-node" to denote a physical node (*i.e.* server or peer) in our system.

2.1 Data Structure

Super-leaf P-nodes within the same data-center are grouped to form a *super-leaf*. Each super-leaf contains between l and $2l$ p-nodes, where l is an operating parameter. Within a super-leaf, we use a gossip-style (or epidemic) algorithm [4] to propagate information held by any subset of p-nodes in a super-leaf to all others. These algorithms are extremely tolerant to link and p-node failures, take $\mathcal{O}(\ln l)$ time and use $\mathcal{O}(l \ln l)$ messages: a super-leaf can therefore be viewed as an atomic unit with a self-consistent view of the world. In the sequel, we therefore refer to the super-leaf when we mean to refer to the collection of p-nodes constituting the super-leaf. For example, we may say that super-leaf F has state s, to mean that all p-nodes of super-leaf F have state s.

Virtual Tree and Partial Representation Overall, super-leaves are organized to form a *virtual tree*, where each tree node has degree between b and $2b$ and each leaf corresponds to a super-leaf [1] Each tree node

[1]From now on, we use the terms leaf and super-leaf interchangeably.

is emulated by every one of its descendent (super)-leaves, that is, p-nodes emulate the nodes on the path from their super-leaf to the tree root. This is similar in principle to the design of Willow [17], except that, unlike Willow, we do not require the tree height to be exactly 129.

Note that a p-node only maintains a *partial* representation of the virtual tree, storing only information about its super-leaf, its ancestors, and the children of these ancestors. (Figure 1.)

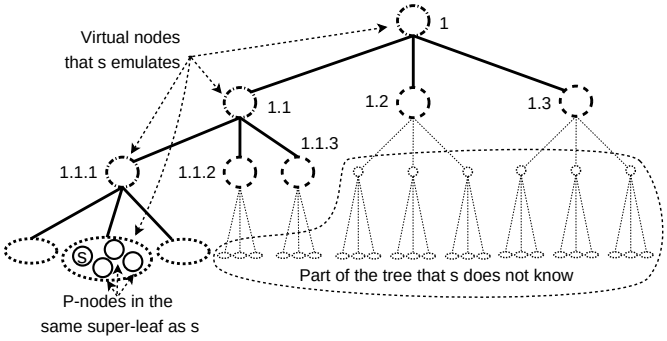


Figure 1. Partial Tree Representation of s

State We gain flexibility in our approach by associating a generic "state" with each tree node (both virtual and real). The value of this state depends on the purpose to which we are using the infrastructure. For simple operations, such as finding the sum of the values held at the leaf nodes, the state corresponds to a partial sum. For more complex operations, such as a content-based filtering of the data held at the leaf nodes, the state would correspond to partial results of the filtering operation.

To provide fault tolerance, we use the following replication rule: all p-nodes that are descendants of a virtual node are required to emulate it, by maintaining an identical copy of that virtual node's state. As we will see later, for efficiency in computation, we require that the state of a virtual node can be deterministically derived from the state of its children. As an example, suppose we are using LOT to compute the sum of the values at the leaves. Then, the state of a node could correspond to a partial sum of its descendant's values. All descendants of the virtual node would therefore carry identical copies of this partial sum. Alternatively, we can think of a virtual node as being associated with an SQL query, whose evaluation requires the aggregation of query evaluations on its children. Our replication rule would require all descendants of the virtual node to maintain copies of the partial results. In the next section, we explain how to efficiently update a virtual node's state while maintaining consistency among replicas.

2.2 Maintenance Algorithm

The state of a virtual node can only change if there is a change in the values held at one of its descendant leaves. Let's say one of the leaf values changes at some time t. This change is not immediately propagated up the tree. Instead, the state of the interior tree nodes is recomputed periodically, using the values at its descendant super-leaves at the time the computation starts.

The algorithm works in rounds. During round i, the state of the height i nodes is recomputed. More precisely, during round i, each super-leaf learns the state of the relevant nodes of height $i-1$, then each super-leaf independently computes the state of the height i node it emulates, namely, its height i ancestor. Because the computation is deterministic and all super-leaves have the same input, all super-leaves emulating a node will obtain the same state. Note that we assume the availability of an approximate global clock so that all p-nodes run the same round of the algorithm simultaneously. Such a clock is easily available these days due to GPS. Slower p-nodes can easily re-synchronize when receiving "early" requests.

Before detailing the algorithm, we describe some of the tuning parameters: ρ is a user chosen reliability parameter and should be at most polylogarithmic in the network size n. The branching factor b is chosen by the user. We suggest $b = \ln^{\lambda} n$ for some constant $\lambda \geq 0$, which may or may not be larger than 1. The algorithm uses parameters $k_1 = 2$ and $k_2 = \frac{1+\rho}{2} \ln n$. Finally, the super-leaf size is constrained between l and $2l$. Set $l = \alpha \cdot k_2$ for some user chosen $1 \leq \alpha \leq 2b$. We suggest taking $\alpha = 2b$. The results of Section 2.4 explain and justify our choices.

The round i of the algorithm is as follow. Each super-leaf assigns k_1 distinct super-leaf members to each one of the children of its height i ancestor.[2] Each designated p-node tries to contact *one*[3] random representative of the child it was assigned to, and learns its state. We assume the existence of an *emulation table* to map from a virtual node to a set of IP addresses of p-nodes who emulate that node: the computation and maintenance of the emulation table is described in Section 2.3. We set a small time-out to guard against slow p-nodes slowing down the whole computation.

[2]The k_1 p-nodes contacting child c_i are distinct. However, the same p-node may be assigned to several (typically l/bk_1) children if there are not enough p-nodes in the super-leaf.

[3]Each p-node only gets one try.

Then, the p-nodes from the super-leaf run a gossip (epidemic) broadcast within the super-leaf to share the information they have learnt. They also discover if, due to failures, the states of some children have not been learnt. In this case, the whole procedure is repeated, this time with k_2 p-nodes assigned to each missing child. At this point, with high probability, all p-nodes know the state of each of the children of the height i ancestor. Finally, each p-node computes the height i ancestor's state.

2.3 Emulation Tables

It turns out that we can use LOT to maintain its own emulation tables. Suppose we define the state of a virtual node u to be *a sample of p-nodes emulating* u.[4] Moreover, let each p-node s also hold a vector `temp` of p-node IP addresses, initialized to m copies of s's own IP address. The algorithm to maintain the emulation tables is then as follows:

During each intra-super-leaf communication (gossip exchange), p-nodes exchange a random half of their `temp` vectors. During each long distance communication, they swap their `temp` vectors. Recall, each p-node only emulates one virtual node at height i, its height i ancestor a_i. We set $\texttt{state}_s(a_i)$ to the value of $\texttt{temp}_s$ taken at the end of round i.

Another description of this algorithm is the following: consider a virtual node u of height i, and the *concatenation* of all $\texttt{state}_s(u)$ over p-nodes s that emulate u. This concatenation is a random permutation of the IP addresses of p-nodes emulating u. Each emulating p-node therefore returns a different part of the concatenation. This allows each p-node to get a nearly uniform random sample of the set of p-nodes that emulate any virtual node u.

Finally, emulation tables are created out of the partial tree representation $\mathcal{P}$ by adding to each non ancestor node $u \in \mathcal{P}$ a list of p-nodes emulating virtual node u. Consider a super-leaf F and denote by $v_1, v_2 \ldots, v_j$ the height $i-1$ virtual nodes present in F's emulation table, excluding F's height $i-1$ ancestor. During round i, F receives one or more copies of the state of $v_1, \ldots, v_j$. The state of v_j is actually a sample of p-nodes emulating v_j. From several samples of nodes emulating v_j, F randomly creates a new sample for the emulation table. A different presentation, more detailed, of the algorithm can be found in [2].

[4]Note that we had earlier stated that each p-node s emulating a virtual node u has the same copy $\texttt{state}_s(u)$ of u's state. Here, we allow a small divergence from this definition, so that each p-node emulating u has a different random sample of p-nodes emulating u.

2.3.1 Joins

We now describe how a new node joins the system. A joining p-node s sends a request to any existing LOT p-node s'. For each child c_i of the root, p-node s' returns a small set of p-nodes emulating c_i. P-node s selects the closest child c_i and forwards its join request to one of the p-nodes emulating c_i. This process iterates, stepping one level down in the tree each time until s locates the appropriate super-leaf, *i.e.* the one closest to it, that it joins.

Alternatively, the p-nodes can run a localization system like Meridian [19], which directly locates the best super-leaf to join.

2.3.2 Structural changes

At the end of round i, each super-leaf checks whether the branching constraint on its height i ancestor is satisfied. If it is not, during round $i+1$, the super-leaves announce the relevant change to the ancestor (the ancestor splits into two nodes, merging with a contiguous node, as in a normal B-tree). Because all super-leaves have received the same information during round i, all super-leaves will reach the same decision regarding the ancestor. Thus they will all update the tree in the same way and announce the same change during round $i+1$, *without further need to communicate to reach agreement.*

At the end of each maintenance process, each super-leaf gets a fresh emulation table, with a partial tree representation that is consistent with that of all other p-nodes. P-nodes that fail will be removed from the emulation tables by the end of the next maintenance run, ensuring good performance.

Balance Denote by $\mathcal{D}_u$ the set of super-leaves descending from u. These are the super-leaves emulating u. Even though the tree is height-balanced, the number of super-leaves in subtrees of the same height may be different. Consider virtual node u of height i. During round i, some p-nodes (not in $\mathcal{D}_u$) try to learn u's value. Denote by $\text{acq}(u)$ the number of these p-nodes and by $\beta_u = \frac{\text{acq}(u)}{|\mathcal{D}_u|}$ the expected load on the p-nodes of $\mathcal{D}_u$. Let

$$\beta = \max_u \left(\frac{\text{acq}(u)}{|\mathcal{D}_u|} \right)$$

Variable β is the maximum expected load on a p-node. We assume that, for a user-chosen $\eta \geq 0$ (see the bound on ε below):

$$\beta \leq \beta_{\max} = \texttt{max\_load} \cdot e^{-1-(2+\eta)\frac{\ln \texttt{max\_load}}{\texttt{max\_load}}} \approx \frac{\texttt{max\_load}}{e} \quad (1)$$

In other words, each time a virtual node u is such that $\beta_u > \beta_{\max}$, the re-balancing mechansims of the algorithm kicks in and re-balances the tree to restaure the constraint $\beta_u \leq \beta_{\max}$.

2.4 Analysis, Performance and Reliability

2.4.1 Performance

Running Time We now state theorems on the performance of LOT, with the proofs deferred to Appendix A.

Theorem 1. *The running time of the algorithm is* $\Theta((1 + \frac{b}{\alpha \ln b}) \ln n)$*, that is,* $\Theta(\ln n)$ *when* $\alpha = 2b$*.*

Proof. The running time is $\mathcal{O}(h \cdot R)$ where R is the running time of a round and h the height of the virtual tree. We have $\log_{2b} \frac{n}{2l} + 1 \leq h \leq \log_b \frac{n}{l} + 1$, or $h = \Theta\left(\frac{1}{\ln b} \ln \frac{n}{l}\right)$. There are two operations during a round: the acquisition of states and the broadcast within the super-leaf, each of which may be repeated once more if some values are missing. The running time of the broadcast within each super-leaf is $\Theta(\ln l)$. Because each p-node sends one or two messages long distance messages and processes at most 2 `max_load` incoming messages the length of both acquisition phases is upper bounded by $2\,\texttt{max\_load} + 2 = 2 \ln \ln n + 2 \leq 2 \ln l + 2$. We have $R = \Theta(\ln l + \texttt{max\_load}) = \Theta(\ln \ln n) = \Theta(\ln b)$. The running time is then

$$T = h \cdot R = \Theta\left(\frac{1}{\ln b} \ln \frac{n}{l} \cdot \ln b\right) = \Theta(\ln n) \qquad \square$$

2.4.2 Reliability

Assumptions We assume that failures are fail-stop, independent, and distributed uniformly at random among the p-node. We denote by p an upper-bound on the individual probability of failure between two consecutive tree re-computations.

We also assume that p-nodes are provided with a uniform random sample of p-nodes emulating the virtual nodes present in their partial tree representation, (the algorithm we present in Section 2.3 allows us to compute almost uniform samples). Then the algorithm succeeds with high probability even in the presence of p-node failures during computation.

Assuming that p-nodes select uniformly at random which connections to drop when they become too numerous, we model the dropping of incoming connections by an increase bounded by ε in the probability of failures from p. The justification for this and the new modified probability of failure $p'$are described below. The detailed proofs of the theorems can be found in appendix.

Lemma 2. *Leaves dropping connections when their load exceeds* `max_load` *long range messages in a round is equivalent to leaves serving all their connections but with an increase of* ε *in the failure probability of targets, where* $\varepsilon \leq \frac{1}{4 \ln^2 \texttt{max\_load}} \cdot \frac{1}{(\texttt{max\_load})^\eta}$*.*

We set $k_1 = \frac{\ln \ln n}{|\ln(2p+\varepsilon)|}$. Note that $\lim_{n\to\infty} \varepsilon = 0$.

Theorem 3. *The algorithm fails with probability:*

$$p_{fail} = \mathcal{O}\left(\frac{1}{n^{(\gamma-1-\delta)+\gamma\rho}}\right)$$

where $\gamma = |\ln(2(p+\varepsilon))|/2$ *is a parameter depending on the failure probability,* $\delta = \frac{\ln \frac{b}{\alpha(1+\rho)\ln b}}{\ln n}$ *is small, and can be ignored if* $\delta \leq 0$ *which is the case most of the time, including when* $\alpha = 2b$*. Last,* ρ *a user-chosen parameter governing the reliability with* $0 \leq \rho \leq \mathcal{O}(\ln n)$ *being acceptable values. Larger* ρ *leads to larger super-leaves, of size* $l = (1 + \rho) \cdot \alpha \cdot \ln n$

2.4.3 Load on P-nodes and Tree Balance

The virtual tree must be maintained in a way that satisfies the bound on β as defined in Equation 1. Denote by $S(u)$ the number of super-leaves that need to acquire u's value. Note that $\text{acq}(u) \leq S(u) \cdot k_2$ since each super-leaf sends at most $2l/b$ messages to $\mathcal{D}_u$. For the highest virtual nodes in the tree, the bound on $\text{acq}(u)$ is loose by a $k_2/k_1 = (1 + \rho) \ln n / \ln \ln n$ factor.

Theorem 4. *For the virtual nodes* u *high in the tree, that is, such that*

$$S(u) \geq (1 + \rho)|\ln(2(p + \varepsilon))| \frac{\ln^2 n}{2e^2} \qquad (2)$$

we have $acq(u) = \mathcal{O}(S(u) \cdot k_1) = \mathcal{O}(S(u) \cdot \ln \ln n)$

This result holds with very high probability. The probability that some (virtual) node exceeds the bound during the execution of the algorithm is smaller than the algorithm failure probability.

In other words, none of the virtual nodes satisfying constraint (2) will have a load exceeding $\mathcal{O}(k_1 \cdot S(u))$.

Message Complexity The number of messages sent by the algorithm is $\mathcal{O}(n \ln n)$. However, not all messages have the same cost. A more interesting metric is to count the number of long distance messages (as opposed to local messages, sent within the super-leaf).

Theorem 5. *When* $\alpha = 2b$*, the number of non-local messages sent is, with high probability,* $\mathcal{O}(n)$

In fact, each p-node initiates on average $2b/\alpha$ long distance messages.

Theorem 6. *Long distance messages are of size $\mathcal{O}(\mathcal{A}gg)$. Local messages are of size $\mathcal{O}(b \cdot (\mathcal{A}gg + k_2))$ where $\mathcal{A}gg$ denotes the size of messages used to describe the state of a node.*

Note that $\mathcal{A}gg \geq 2m$ because they are two vectors of m elements used for the maintenance of the emulation tables. However, as explained in Section 3.4 in the case of two internal nodes merging, $\mathcal{A}gg = \mathcal{O}(b \cdot m)$; and depending on the implementation, we may always have $\mathcal{A}gg = \mathcal{O}(b \cdot m)$.

3 Distributed Index

We now describe how to use LOT to create a distributed index that supports range queries in a distributed database. We focus here on uni-dimensional ranges, deferring a discussion of multi-dimensional ranges to Section 5.2.

3.1 Index and Data Deployment

In a standard uni-dimensional B-tree, data is placed in sorted order on the leaf nodes, and internal nodes are associated with a range that covers the data values of its descendants. Our approach adds a layer of indirection: each p-node leaf node can store data in more than one range. However, it is responsible for indexing a range of data values stored at other p-nodes. It is these indices that are contiguous, and constitute the 'state' held in a LOT node. For instance, a p-node may hold data values 1, 5, and 10, and be responsible for the range [6-9]. Then, it will have pointers to every p-node that holds data values in the range [6-9], and is pointed to by p-nodes responsible for the ranges that include the values 1, 5, and 10. Nodes at higher levels in the tree use their state variables to aggregate these ranges in the normal fashion. For simplicity, we assume that the range of an internal node is divided between its children with no gaps and no overlaps.

Recall that we cluster p-nodes to form super-leaves. We therefore aggregate the index ranges at the p-nodes to form the index range of the super-leaf and replicate the index range of the super-leaf at every p-node in the super-leaf (Figure 2).

3.2 Range Query Processing

Conceptually, range queries are issued at the root node, and recursively forwarded down the tree to the appropriate nodes. In practice, the nodes are virtual and the request is forwarded to a randomly chosen p-node emulating that node (Algorithm 1 where r is the querying range and i the level initialized with the height of the tree).

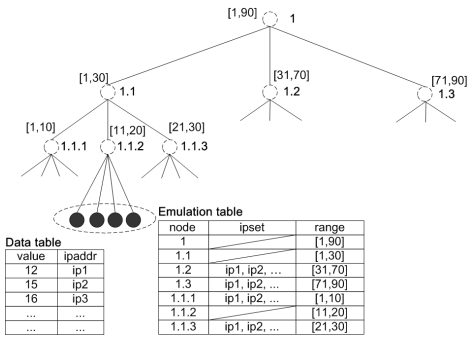


Figure 2. Range Index and Data Deployment

Algorithm 1 *range_query_processing*(r, i)

```
1:  if i==0 then
2:    Use the data table to return the p-node IP's of entries
      matching the range.
3:  else
4:    for each routing entry at level i intersecting with r do
5:      [Denote by C the node, by r_C the resulting intersection]
6:      if C is our ancestor (of height i) then
7:        call (locally) this.range_query_processing(r_C, i − 1)
8:      else
9:        get a random p-node s_C emulating C
10:       call (remotely) s_C.range_query_processing(r_C, i − 1)
11:     end if
12:   end for
13: end if
```

Note that this algorithm will fail if a p-node fails while forwarding the request. To make the algorithm fault-tolerant, on receiving a query, each p-node can return, not of a single p-node emulating the next hop, but a list of p-nodes emulating the next hop. The enquirer can then propagate the query to more than one of these p-nodes in parallel. In essence, the query execution plan is parallelized among multiple execution paths. As long as any one of these paths succeeds, the overall query will succeed. Furthermore, each temporary result may be cached at the super-leaf to improve performance.

3.3 Joins and Leaves

We now describe how to maintain the index structure when new p-nodes join and leave. Recall that a p-node s always joins the closest super-leaf from a network perspective. At join time, it gets a copy of the super-leaf state while being added to the list of super-leaf members. Then, p-node s issues a data update operation (described in the next paragraph) to get its data inserted in the appropriate set of indices held at other super-leaves. Previously joined p-nodes issue similar operations in the case of data updates. While s will not be included in the emulation tables until the

next maintenance phase, thus not receive any load until then, *its data is available immediately to all queries.*

To perform a data update, p-node s first issues a query to locate the data table (and equivalently the super-leaf) that maintains the range corresponding to the item to insert. Then p-node s sends a notification of data insertion to a p-node of that super-leaf. The message is propagated throughout the super-leaf via a gossip broadcast algorithm to update all copies of the data table.

Note that p-node s's data always resides on s. However, the data may be replicated within s's own super-leaf if desired. Were s to leave or permanently fail, s's super-leaf would notice and issue (several copies of) a delete operation on s's data. The data index entry corresponding to s contains s's address so that multiple delete operations may be issued without damaging the rest of the index.

Note that we have a 'best-effort' semantics regarding consistency: during the gossip-based propagation of a data update, the data table replicas are inconsistent, thus the same query being answered by different p-nodes may return different results. However these inconsistencies are quickly fixed at the next index maintenance operation. Alternatively, if strong consistency is desired, traditional transaction control mechanisms (write locking) may be applied to the super-leaf. This is not prohibitively expensive thanks to the rather small super-leaf size and the low latency of communications within the super-leaf.

3.4 Index Maintenance

As p-nodes join and leave, super-leaves will acquire members who are not part of the emulation tables at other p-nodes. Hence, they will not serve any query load. New p-nodes are incorporated into the emulation tables during the maintenance phase, which proceeds as described in the "membership" algorithm in Section 2.3. At the end of the maintenance phase, all p-nodes known to the system at the start of the maintenance phase can start serving query load.

In addition to maintenance of the emulation tables, super-leaves may need to be re-balanced if too many p-nodes join them. This is done using the split technique described in Section 2.3. If a super-leaf split causes an internal node to split, then the range at the internal node is partitioned into two ranges, which are then associated with the two new internal nodes.

The case of two internal nodes merging, and re-balancing operations, are complex and not particularly illuminating. We describe them next.

Essentially, a re-balancing operation is the transfer of a child of node u to node v where v is a node immediately to the left of right of u. A merge between u and v is a special, extreme case of re-balancing, one in which all children of u are transferred to v. Further, re-balancing operations between u and v have no consequences on other nodes except the change in index range of u and v. This allows the tree to efficiently re-balance virtual nodes in case of data skew, but also in the case of load skew.

During round i, information regarding the children of the height i ancestor is being exchanged. Information about children of these children is required for a tree-merge. The approach in the full-scale simulator we describe in Section 4.2 is to always send this information, at the cost of larger messages, but with the gained flexibility of being able to do B-tree branch re-balancing to re-balance branches and indices. Alternatively, this information about the children of the children may be sent/request only when needed, for a merge. This makes for smaller messages but a substantially more difficult implementation and was done in our leight-weight simulator of Section 4.1

3.5 Query Processing Performance

The number of hops needed to answer a single point query is $h+1$ where $h \leq \log_b \frac{n}{l} + 1$ is the height of the tree. The extra message is needed to reach the data itself after reaching the index table at the super-leaf. Assuming failures and emulating tables to be uniformly random, each query will stumble on an expected $E = hp/(1-p)$ failed p-nodes before reaching destination.

Thanks to the high branching factor of our B-tree-like structure, the tree hight is smaller than that of other overlays using binary trees [8, 7], thus improving query performance. Furthermore, this number of hops needed to answer a query is constant, regardless of failures. This is of prime importance in a dynamic environment where churn can be high. In contrast, if a query is routed *around* failed physical nodes, the number of (live) hops needed to complete a query increases with the number of failed nodes, thus leading to performance issues under churn. The simulations of [6] (and we expect the same to hold for [8, 7]) suggests that the number of hops per query grows quadratically with the failure rate.

4 Evaluation

In this section, we present an experimental evaluation of LOT, as well as a direct comparison with the BATON/VBI family of algorithms.

4.1 Reliability

We developed a light-weight, highly scalable simulator specifically to evaluate the robustness of the LOT framework, especially at large scale.

To measure robustness, after growing the network until reaching a system of 10,000 p-nodes, we randomly kill a fraction p of the p-nodes, then run the maintenance algorithm. Recall that during the maintenance phase, membership tables are updated using two rounds of inter-cluster messages. In Figure 3 we plot the fraction of these membership updates that fail, as a function of the failure rate, and for two values of m. Note that even with a failure rate of 20%, no membership updates fail! Membership updates start to fail as the p-node failure rate increases beyond 20%, with a rate of increase smaller for larger m. In practice, we do not expect more than about 5% of the nodes to fail between two runs of the maintenance algorithm. Therefore, even with these high churn rates, we see that LOT performs robustly.

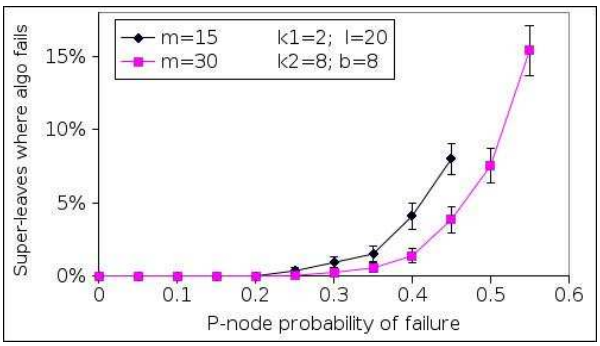


Figure 3. Reliability

4.2 Experimental Setup

To study other aspects of LOT, and to compare it with the BATON/VBI family, we used the discrete event simulator P2Psim[5]. We also implemented our own version of the BATON/VBI algorithms based on the source code we obtained from the authors, and with small modifications (described later).

We generate a data center topology by employing a placement strategy similar to that developed in Brite [11]. Here, we assume that all data centers are distributed in a square area with a side equivalent to 100ms of delay. We divide this square into 16 non-overlapping smaller squares, with one data center per square. The number of p-nodes per data center follows a power-law distribution with exponent 2.3 The p-nodes are placed uniformly at random within each small square. The latency between two p-nodes is then modeled by their Euclidean distance. This results in low delays between p-nodes within the same data center, with larger delays between p-nodes in different data centers. Note that we are being conservative, in that the ratio of delays in practice is likely to be about 200:1, rather than the 4:1 in our simulations.

The system starts with a single p-node. Then p-nodes join the system according to a Poisson distribution. We varied the join rate, having between 1000 and 5000 p-nodes join over the course of the experiment. Our experimental data has been measured after the system stabilizes, and we report the average number of p-nodes in the system. We set $l = 15$, $b = 4$ and $m = 20$. To realistically simulate node churn, the life time of p-nodes follows a power-law distribution with exponent $\alpha = 0.9$. This matches the Gnutella host life time distribution reported in [15].

In this work, we focused on uni-dimensional data and implemented the same framework as in [8] Each p-node carries an integer randomly chosen from the range $[1, 10\,000\,000]$. Range queries are generated in the following way. The range starting point is randomly chosen from the data range; the range span is computed by choosing a random integer from $[1, 10]$ and multiplying it with the expected range span held by a super-leaf. Queries are issued by randomly chosen live p-nodes at a random instant during the measurement time.

4.3 Performance Evaluation

4.3.1 Range Query Processing and Join Delay

We issued 10,000 range queries and measured the latency (Figure 4). As expected, the latency increases logarithmically with the system size. The average cost in number of messages per query was only slightly larger than the height of the tree. This is despite queries that cover multiple super-leaves. Note, we only report the time to locate the p-nodes holding the requested data: we did not include the time to actually retrieve the data itself, because this depends on the size of the data itself and is independent of the query cost.

We also measured the latency of a join request, from the point of view of the joining p-node. This excludes the propagation time of the join request within the joined super-leaf. Results were very similar to those of Figure 4, which is to be expected because the process of a p-node joining is very similar to a single point query. We do not show the graph to conserve space.

[5]P2Psim: http://pdos.csail.mit.edu/p2psim/

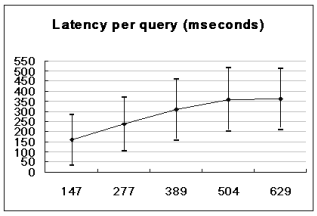


Figure 4. Range Query Latency

4.3.2 Maintenance Costs

We now turn to the maintenance phase and show that it both short and light-weight. Table 1 shows the total latency for the execution of a complete maintenance phase. Note that even with several hundred nodes, the entire process finishes in a few seconds.

Nb of P-nodes	132	264	398	523	635
Average delay (ms)	1116	1846	1923	3268	3622

Table 1. Maintenance Phase Duration

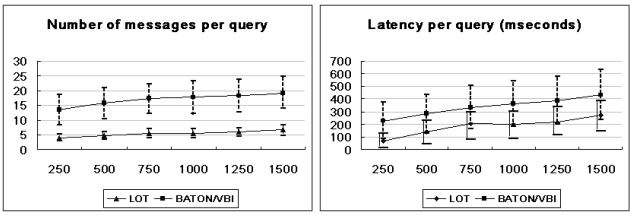
An additional metric of interest is the bandwidth consumed by inter super-leaf messages. This is plotted in Figure 5 (b). We mentioned in Section 2.4 that the number of long distance messages sent by each p-node is a small constant. This can be verified with Figure 5 (a). Figures 5 (c) and 5 (d) illustrate our design principle to trade off long distance messages for local ones: we see that the number of intra-super-leaf messages is a small fraction of the number of inter-super-leaf messages.

4.4 Performance Comparison

We now compare the query processing cost of LOT to BATON and VBI [8, 7], two balanced tree overlays used for range query processing. To make the comparison fair, we re-implemented these systems under the P2Psim simulator. While our intent was to compare the algorithms under a realistic scenario, we found out that neither work addresses the concurrency issues that arise in a high-churn environment. We therefore enhanced BATON/VBI with a simple concurrency control, locking p-nodes when they update their routing and range index. To avoid dead-lock issues, p-nodes fail when encountering a locked node, and retry the operation later. They eventually give up after a few unsuccessful attempts.

Unfortunately, under our original scenario of Section 4.2, failures during the join process rise to dramatically high levels (above 50 %), making it impossible to build a large-scale BATON and VBI network. Consequently, we reverted to comparing LOT with these systems in a nearly static environment, with systems of size between 250 and 1500.

We measured both the number of messages used per range query and the query latency (Figure 6). The results show a statistically significant performance advantage for LOT over BATON/VBI. Concurrent to this work, the authors of BATON and VBI published BATON* [6] addressing the query performance issue by increasing the fan-out of the tree. The number of messages per query appears to be competitive with that of LOT. However, BATON* also does not address the issue of concurrency, making the system impracticable in high-churn environments.



(a) (b)

Figure 6. Query Processing Cost Comparison

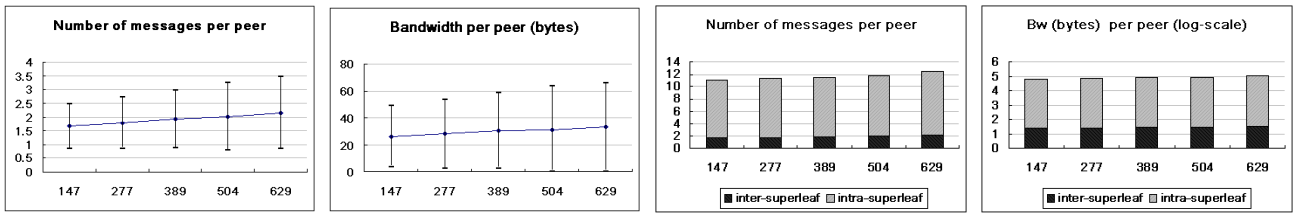
5 Extending State Information for Other Applications

As mentioned in Section 1.1, flexible state definition is one of the LOT design principles that allows use of the LOT framework for a variety of applications. In this section, we show how to use other state definitions to enable two important applications: in-network computation and multi-dimensional indexing.

5.1 In-Network Computation

In-network computation tries to compute an aggregate of the values distributed over the network without sending all the values to a central computation node. Practically any in-network computation, such as computing aggregates like sum of the super-leaf values, can be carried out in LOT by a simple definition of the appropriate 'state' value. Other in-network computation algorithms are well known in the literature, such as [10] and [9]. Like LOT they all require $\mathcal{O}(\ln n)$ time.

In terms of the number of messages, [9] is superior to LOT by a factor of $\mathcal{O}(\ln n / \ln \ln n)$. However, we believe that the appropriate cost metric is the number of long distance messages, which is $\mathcal{O}(n)$ in both cases. Note that if a node of [10] or a leader of [9] fails during the computation, the result will almost certainly be *incorrect.* This is because these algorithms tolerate



(a) (b) (c) (d)

Figure 5. Maintenance Cost Per P-node

link failures but not node failures. In contrast, our algorithm tolerates both.

In the context of P2P databases, we expect a considerable degree of node churn, either due to node failure, or because the owner of the node has decided to remove it from the system. When used in such environments, systems that do not tolerate node churn are inapplicable. We believe that LOT is an attractive solution in such environments.

5.2 Multi-dimensional Indexing

Extending the state of LOT nodes to contain multi-dimensional ranges instead of one-dimensional intervals is orthogonal to the design and structure of LOT. Basically, treating LOT as a R-tree like structure is a straightforward extension of this paper. Multi-dimensional bounding regions, however, have implications on the query processing algorithm since multiple ranges can satisfy a point or a range query (because of overlapping ranges). Exploiting parallelism in LOT allows for pursuing all paths that are consistent with a range query at the same time.

6 Related Work

Our work is in the general area of peer-to-peer Distributed Hash Tables (DHTs) and the related area of tree-overlays for query processing in P2P distributed databases. The literature in both areas is extensive, therefore, for considerations of space, we discuss only the work that is closest to ours.

With respect to the DHT literature, our work is similar in spirit to the Astrolabe[18], Willow [17], PIER [5], and SDIMS [20] DHTs. Astrolabe [18] organizes peers into a tree. Like LOT, each non-leaf node maintains a user-defined state table, and in-network aggregates are computed in multiple rounds. To gain fault tolerance, each non-leaf node is monitored for failure, and, on failure detection, an alternative is elected. Astrolabe differs from LOT in two ways: (1) it uses a physical tree instead of a virtual tree and (2) it gains reliability through explicit leader election instead of virtualization and replication.

The Willow [17] overlay binary tree supports aggregate query processing, multicast, and publish/subscribe. Like LOT, internal tree nodes aggregate the values of children nodes, and, in turn are emulated by all their descendants. It differs from LOT in: (1) the tree is binary and of fixed height, so is less flexible than a B-tree (2) even with a few p-nodes, every computation requires 129 rounds (3) it ignores physical locality and (4) it handles consistency with only best-effort semantics, while LOT can guarantee a stronger consistency by periodically running a membership maintenance procedure.

PIER [5] is a P2P query processing system that handles a variety of SQL queries in P2P networks, including range queries, using several overlay network protocols. For range query processing, PIER uses a distributed prefix tree [13], which can get unbalanced, as discussed below.

SDIMS [20] system is an information management system that targets aggregation queries in large-scale distributed systems. It achieves scalability by using Plaxton-network [12] DHT to arrange peers in scalable aggregation trees. Although SDIMS has many good properties, it does not explicitly take data center topology into account, so long-range links are as likely to be used as local links. Second, it only provides eventual consistency, compared to the stronger consistency models possible in LOT. Finally, although it employs replication to improve robustness, it does not guarantee a failure probability bound, which is a necessity in a high-churn environment.

With respect to the P2P database literature, our work can be viewed as a natural extension to the work on tree-overlay-based range query processing as proposed in P-Grid [1], Prefix Hash Tree [13], P-trees [3], and BATON, VBI, and BATON* [8, 7, 6].

P-Grid [1] is based on a randomized binary prefix tree. However, the prefix tree becomes unbalanced under skewed data distribution so that the worst-case search cost cannot be logarithmically bounded. A similar prefix-tree based overlay network in described

in [13] and shares the same problem as P-Grid. In contrast, LOT builds a balanced tree overlay network regardless of the data distribution, and the worst-case query processing cost is guaranteed to be logarithmic in the network size. P-tree [3] uses a B+-tree index to support P2P range query processing. However, it relies on an underlying protocol (*e.g.*, Chord [16]) to deal with query routing and overlay network maintenance, while we include these as part of LOT's design. This allows us to reuse LOT itself to do membership and routing table maintenance. Second, P-tree can only achieve eventual consistency through running a stabilization process individually on each peer, which can result in performance degradation. In contrast, we periodically run an inexpensive maintenance algorithm to synchronize and achieve a strong performance guarantee.

Jagadish *et al.* have proposed a series of tree-based overlay network protocols that deal with range query processing [8, 7, 6]. BATON [8] uses a balanced binary search tree with in-level links for efficiency, fault-tolerance, and load-balancing. VBI [7] enhances BATON with a limited degree of virtualization and focuses on employing multi-dimensional indexes to support more complex range query processing. Finally, BATON* [6] speeds up the query processing by increasing tree fan-out. These systems share the following common problems. First, their tree overlay structures fix the tree fan-out. So, each peer join or leave can cause a tree structural change. This lacks the resilience of a B-tree. Second, they employ an in-place synchronization strategy that globally updates routing and range index information immediately after any change. Under a high-churn environment, this strategy either requires a prohibitively expensive mutual exclusion mechanism to guarantee the consistency and performance, or tolerates considerable inconsistency. These problems are addressed in LOT through the employment of B-tree topology and a light-weight periodic maintenance algorithm.

7 Conclusion

In this paper, we presented LOT, a virtual B-tree like structure that is robust, load-balanced, efficient and able to handle high churn situations. As an example of LOT's fitness to highly distributed peer-to-peer environments, we show how to efficiently answer range queries using LOT. Our simulations demonstrate the effectiveness of our design and its superiority over previous work.

Our contribution lies in the unique combination of four design principles: proximity-based clustering, node virtualization, path and node replication, and flexible state definition, which correspond to the four major strengths of LOT: performance, load-balance, robustness and extensibility.

Acknowledgement: We would like to thank H. V. Jagadish and Q. H. Vu for providing the code and explanations of BATON and VBI.

References

[1] K. Aberer. P-Grid: A self-organizing access structure for P2P information systems. In *6th CooplS*, 2001.

[2] A. Allavena and S. Keshav. Lot: Fast, efficient and robust in-network computation. Technical Report cs-2006-22, University of Waterloo, 2006.

[3] A. Crainiceanu, P. Linga, J. Gehrke, and J. Shanmugasundaram. Querying peer-to-peer networks using p-trees. In *Proc. 7th WebDB*, 2004.

[4] U. Feige, D. Peleg, P. Raghavan, and E. Upfal. Randomized broadcast in networks. *Random Structures and Algorithms*, 1990.

[5] R. Huebsch, . M. Hellerstein, N. Lanham, B. T. Loo, S. Shenker, and I. Stoica. Querying the internet with PIER. In *Proc. 29th VLDB*, 2003.

[6] H.V.Jagadish, B. C. Ooi, K. L. Tan, Q. H. Vu, and R. Zhang. Speeding up search in peer-to-peer networks with a multi-way tree structure. In *Proc. 25th ACM SIGMOD*, 2006.

[7] H. V. Jagadish, B. Ooi, Q. Vu, and A. Z. R. Zhang. VBI-tree: A peer-to-peer framework for supporting multi-dimensional indexing schemes. In *Proc. 21st IEEE ICDE*, 2005.

[8] H. V. Jagadish, B. C. Ooi, and Q. H. Vu. Baton: A balanced tree structure for peer-to-peer networks. In *Proc. 31st VLDB*, 2005.

[9] S. Kashyap, S. Deb, K. Naidu, R. Rastogi, and A. Srinivasan. Efficient gossip-based aggregate computation. In *Proc. 25th ACM PODS*, 2006.

[10] D. Kempe, A. Dobra, and J. Gehrke. Computing aggregate information using gossip. In *Proc. 44th IEEE FOCS*, 2003.

[11] A. Medina, A. Lakhina, I. Matta, and J. W. Byers. Brite: An approach to universal topology generation. In *9th IEEE MASCOTS*, 2001.

[12] C. G. Plaxton, R. Rajaraman, and A. W. Richa. Accessing nearby copies of replicated objects in a distributed environment. In *Proc. 9th ACM SPAA*, 1997.

[13] S. Ramabhadran, S. Ratnasamy, J. M. Hellerstein, and S. Shenker. Brief announcement: prefix hash tree. In *Proc. 23rd ACM PODC*, 2004.

[14] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Shenker. A scalable content-addressable network. In *Proc. SIGCOMM*, 2001.

[15] S. Saroiu, P. K. Gummadi, and S. D. Gribble. A measurement study of peer-to-peer file sharing systems. In *Proc. of MMCN*, 2002.

[16] I. Stoica, R. Morris, D. Karger, F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proc. SIGCOMM*, 2001.

[17] R. v. Renesse and A. Bozdog. Willow: DHT, aggregation, and publish/subscribe in one protocol. In *IPTPS04*, 2004.

[18] R. van Renesse, K. Birman, and W. Vogels. Astrolabe: A robust and scalable technology for distributed system monitoring, management, and data mining. *ACM TOCS*, 21(2), May 2003.

[19] B. Wong, A. Slivkins, and G. Sirer. Meridian: A lightweight network location service without virtual coordinates. In *Proc. SIGCOMM*, 2005.

[20] P. Yalagandula and M. Dahlin. A scalable distributed information management system. In *Proc. SIGCOMM*, 2004.

A Proofs of Section 2.4

Proof of lemma 2. First note that the p-nodes whose connections are dropped are chosen uniformly at random:

Let H be a descendent p-node of a virtual node u. Consider the set of p-nodes that are set to learn u's value during the round and assume that H will drop some connections. Break the process of p-nodes selecting their communication partner uniformly at random into two steps. First, an oracle chooses the histogram distribution of the load of the potential partners and randomly attaches a p-node to each load. (The probability distribution of the histograms is the same as if p-nodes selected their targets uniformly at random.) Then, each p-node selects at random the load of its communication partner – the probability is proportional to the load – and attempts to communicate with the p-node responsible for that load. This process is the same as if p-nodes chose their communication partner uniformly at random

Thus it is equivalent to consider that p-node targets fail uniformly at random with probability $p' = p + \varepsilon$ where ε is the probability that a connection is dropped. We now compute an upper-bound on ε.

Consider p-node f, descendent of virtual node u. Denote by $s = |\mathcal{D}_u|$ the number of p-nodes that emulate u and denote by m the number of p-nodes that try to learn u's value during the current round. By our assumption, each one of these m p-nodes selects a p-node uniformly at random in $\mathcal{D}_u$. The probability that p-node f has received j connections is therefore $\binom{m}{j}\left(\frac{1}{s}\right)^j\left(1-\frac{1}{s}\right)^{m-j}$. Denote by E the expected number of connections that f drops. The probability for a given p-node to see its connection dropped is $\varepsilon = \frac{sE}{s} = E$. We have

$$\begin{aligned}
\varepsilon &= \sum_{j=\texttt{max_load}}^{m} (j-\texttt{max_load})\cdot\binom{m}{j}\left(\frac{1}{s}\right)^j\left(1-\frac{1}{s}\right)^{m-j} \\
&\le \sum_{j=\texttt{max_load}}^{m} (j-\texttt{max_load})\cdot\left(\frac{m\cdot e}{j}\right)^j\left(\frac{1}{s}\right)^j\cdot 1 \\
&\qquad\qquad \text{since } \tbinom{m}{j} \le \left(\tfrac{m\cdot e}{j}\right)^j \\
&\le \sum_{j=\texttt{max_load}}^{m} (j-\texttt{max_load})\cdot\left(\frac{\alpha\cdot e}{j}\right)^j \\
&\qquad\qquad \tfrac{m}{s}\le\alpha \text{ by definition of } \alpha \\
&\le \sum_{j=\texttt{max_load}}^{\infty} (j-\texttt{max_load})\cdot\left(\frac{\texttt{max_load}}{j}\cdot e^{-(2+\eta)\frac{\ln \texttt{max_load}}{\texttt{max_load}}}\right)^j \\
&\qquad\qquad \text{bound on } \alpha \\
&\le \sum_{\kappa=0}^{\infty} \kappa\cdot 1\cdot e^{-(\kappa+\texttt{max_load})(2+\eta)\frac{\ln \texttt{max_load}}{\texttt{max_load}}} \quad \texttt{max_load}\le j \\
&\le e^{-(2+\eta)\ln \texttt{max_load}}\int_0^{\infty} x e^{-x(2+\eta)\frac{\ln \texttt{max_load}}{\texttt{max_load}}}dx \\
&= \left(\frac{1}{\texttt{max_load}}\right)^{2+\eta}\cdot\left(\frac{\texttt{max_load}}{(2+\eta)\ln \texttt{max_load}}\right)^2 \\
&\qquad\qquad \textstyle\int_0^{\infty} x e^{-\beta x}dx = \beta^{-2} \\
&\le \frac{1}{4\ln^2 \texttt{max_load}}\cdot\frac{1}{(\texttt{max_load})^{\eta}}
\end{aligned}$$

□

Proof of Theorem 3. For the purpose of an upper-bound on the algorithm failure probability, we may ignore the successes of first phase acquisitions and only consider the second phases. Recall, p is the probability that a p-node is dead. Let $p' = p+2\varepsilon$. We justify below that p' is an upper-bound on the probability that a request to a p-node is not answered during the second phase, either because the p-node is dead, or because the p-node is overloaded.

Some p-nodes may receive `max_load` or more in-coming messages during the first acquisition phase. These p-nodes may be unwilling to serve up to `max_load` requests during the second acquisition phase because they may exceed their load quota of `max_load` connections for each phase. We do not assume that p-nodes are able to distinguish between first and second phase acquisition messages, therefore they may serve too much during the first phase. Since these p-nodes may fail to answer requests during the second acquisition, we mark them as dead. Although the number of these p-nodes may be computed, for convenience we shall use $\varepsilon|\mathcal{D}_u|$ as a rough upper-bound, yielding an increase of ε in the failure probability of targets. Applying Lemma 2 yields another increase, by ε as well, of the probability

of targets failing to answer a query.

The probability that a p-node set to acquire the value from a given sub-tree T succeeds is at least $(1-p)\cdot(1-p')$. The probability that a super-leaf fails to acquire the value of T is at most $(1-(1-p)(1-p'))^{k_2} \leq (p+p')^{k_2}$ when there are $k_2 = \frac{(1+\rho)}{2}\ln n$ p-nodes set to acquire the value from T.

Taking a union bound on all branches whose value a super-leaf needs to acquire, for all super-leaves and for all rounds, we get an upper-bound on the failure probability of the algorithm: $p_{\text{fail}} \leq \left(2b\frac{n}{l}h(p+p')^{k_2}\right)$. Note that $bh/l \leq b/(\alpha(1+\rho)\ln b)$. We have

$$\begin{aligned} p_{\text{fail}} &\leq 2e^{\ln n(1+\frac{\ln\frac{b}{\alpha(1+\rho)\ln b}}{\ln n})-\frac{(1+\rho)\ln n|\ln(2(p+\varepsilon))|}{2}} \\ &= \frac{2}{n^{\frac{|\ln(2(p+\varepsilon))|}{2}-1-\delta}} \cdot \frac{1}{\left(n^{|\ln(2(p+\varepsilon))|}\right)^{\frac{\rho}{2}}} \end{aligned}$$

with

$$\delta = \frac{\ln\frac{b}{\alpha(1+\rho)\ln b}}{\ln n}$$ □

Proof of Theorem 4. Consider a virtual node u. There are $S(u)$ super-leaves trying to acquire u's value. Each of them fails to do so during the first phase with a probability upper-bounded by $1/\ln n$ (see proof of Theorem 5). Denote by M the number of super-leaves that need to run a second acquisition phase and by μ the expected value of M. Bound M using a Chernoff bound:

$$\Pr[M \geq (1+\delta)\mu] \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu \leq \left(\frac{e}{1+\delta}\right)^{(1+\delta)\mu}$$

Note that $\mu \leq S(u)/\ln n$. Set $(1+\delta)\mu = e^2 S(u)/\ln n$, which guarantees that $\frac{e}{1+\delta} \leq \frac{1}{e}$, and we have

$$P_1 = \Pr[M \geq \frac{e^2 S(u)}{\ln n}] \leq e^{-\frac{e^2 S(u)}{\ln n}}$$

Define P_1 to be the probability of exceeding this bound. We want P_1 small enough that the probability of some vnode exceeding the bound is smaller than P_{fail}, the algorithm failure probability. There are at most $2n/lb$ vnodes satisfying requirement (2). It is sufficient to have $P_1 2n/lb \leq P_{\text{fail}}$, which can be rewritten as

$$e^{-\frac{e^2 S(u)}{\ln n}} * 2n \leq 2ne^{-\frac{(1+\rho)\ln n|\ln(2(p+\varepsilon))|}{2}}$$

equivalent to Equation 2. This shows that with sufficiently high probability, all super-leaves learn the subtree value during the first acquisition phase, that is, $\text{acq}(u) = k_1 S(u)$ □

Proof of Theorem 5. The number N_1 of messages sent during all first acquisition phases is $\mathcal{O}(\frac{n}{l}2bhk_1)$ where $k_1 = \ln\ln n/|\ln(2p+\varepsilon)|$ is the number of p-nodes that are dedicated to any given subtree. We simplfy the expression $\mathcal{O}(\frac{n}{l}2bhk_1)$ by substituing by their values $l = \alpha\cdot b$ and $h = \frac{\ln n}{\ln b} = \frac{\ln n}{\lambda\ln\ln n}$. We get

$$\begin{aligned} N_1 &= \mathcal{O}(\frac{b}{l}\cdot k_1 \cdot h \cdot n) \\ &= \mathcal{O}(\frac{b}{\alpha k_2}\cdot \ln\ln n\frac{\ln n}{\ln\ln n}\cdot n) \\ &= \mathcal{O}(\frac{b}{\alpha}\cdot\frac{\ln n}{(1+\rho)\ln n}\cdot n) \\ N_1 &= \mathcal{O}(\frac{b}{\alpha}\cdot\frac{1}{1+\rho}\cdot n) \end{aligned}$$

A second phase acquisition is run only if the first phase one is unsuccessful. A first phase acquisition fails with probability upper-bounded by $(2p+\varepsilon)^{k_1} = 1/\ln n$. We use a Chernoff bound to bound the number M of these unsuccessful first phase acquisitions:

$$\Pr[M \geq (1+\delta)\mu] \leq \left(\frac{e^\delta}{(1+\delta)^{1+\delta}}\right)^\mu \leq \left(\frac{e}{1+\delta}\right)^{(1+\delta)\mu}$$

We have $\mu \leq 2bhn/(l\cdot\ln n) \leq 2n/((1+\rho)\ln n)$. Set $(1+\delta)\mu = 2e^2 n/((1+\rho)\ln n)$, which guarantees that $\frac{e}{1+\delta} \leq \frac{1}{e}$, and we have

$$\Pr[M \geq \frac{2e^2 n}{(1+\rho)\ln n}] \leq e^{-\frac{n}{(1+\rho)\ln n}} \leq e^{-n/2}$$

The last inequality holds for sufficiently large n.

With high probability, the number N_2 of messages sent during all second phase acquisitions is upper-bounded by $\frac{2e^2 n}{(1+\rho)\ln n}\cdot\frac{2l}{b} \leq \frac{2e^2 n}{(1+\rho)\ln n}\cdot 4k_2$, that is

$$N_2 = \mathcal{O}(n)$$

Finally, the total number N of messages sent to non-(super-leaf)-local p-nodes is $N = N_1 + N_2$:

$$N = \mathcal{O}\left((1+\frac{b}{\alpha(1+\rho)})\cdot n\right)$$ □

Proof of Theorem 6. Long distance messages are of size $\mathcal{O}(\mathcal{A}gg)$ since each long distance message contains the information about the state of a single node.

There are at most $2b-1$ virtual nodes whose state needs to be exchanged within the superleaf. There are also messages regarding the assignement of servers to virtual child for the next round of acquisitions. More precisely, this is an assignement of k_2 p-nodes to each children, for a total of at most $(2b+1)k_2$. Hence the size of local messages is $\mathcal{O}(b\cdot(\mathcal{A}gg + k_2))$. □