Implementing Aggregation and Broadcast over Distributed Hash Tables*

Ji Li
jli@csail.mit.edu

Karen Sollins
sollins@csail.mit.edu

Dah-Yoh Lim
dylim@csail.mit.edu

MIT Computer Science and Artificial Intelligence Laboratory
32 Vassar Street
Cambridge, MA 02139

ABSTRACT

Peer-to-peer (P2P) networks represent an effective way to share information, since there are no central points of failure or bottleneck. However, the flip side to the distributive nature of P2P networks is that it is not trivial to aggregate and broadcast global information efficiently. We believe that this aggregation/broadcast functionality is a fundamental service that should be layered over existing Distributed Hash Tables (DHTs), and in this work, we design a novel algorithm for this purpose. Specifically, we build an aggregation/broadcast tree in a bottom-up fashion by mapping nodes to their parents in the tree with a *parent function*. The particular parent function family we propose allows the efficient construction of multiple interior-node-disjoint trees, thus preventing single points of failure in tree structures. In this way, we provide DHTs with an ability to collect and disseminate information efficiently on a global scale. Simulation results demonstrate that our algorithm is efficient and robust.

Categories and Subject Descriptors

C.2.4 [**Computer-Communication Networks**]: Distributed Systems

General Terms

Algorithms, Performance

Keywords

Peer-to-peer, Distributed Hash Table, Aggregation, Broadcast, Tree

1. INTRODUCTION

In recent years, P2P systems have been widely deployed and used, especially those for file-sharing, such as Napster [1], Gnutella [2], KaZaA [3], and Freenet [4]. We believe that as P2P networks grow in popularity, there is an emerging need to collect and propagate (or even broadcast) aggregate information from such systems on the parts of both administrators and users. For example, in a P2P storage system ([5]) or a Grid-like environment ([6]) it is valuable to learn about aggregate available storage or computation capabilities. In particular, some recently proposed randomized DHT topologies, like Viceroy [7] and Symphony [8], require network size estimates to tune routing performance. Under other circumstances, lifetime distribution and other characteristics may be valuable. Broadcasting can also be used to search for arbitrary semi-structured queries that are not supported by the current DHT lookup. Other applications, such as median distribution, need this functionality, too.

A DHT is a distributed resolution mechanism for P2P systems that manages the distribution of data among a changing set of nodes by mapping keys to nodes. DHTs allow member nodes to efficiently locate stored resources by name without using centralized servers. A large number of DHTs, such as CAN [9], Chord [10], Tapestry [11] and Pastry [12], have been proposed. These systems are expected to eventually become the fundamental components of large scale P2P distributed applications in the near future, and would therefore require the aggregation/broadcast functionality.

In this work, our goal is *to enable a P2P network to compute an aggregation function (MIN, MAX, COUNT, SUM, AVG) over data residing at nodes or over the network itself, and subsequently broadcast information on DHTs.* We believe that a good aggregation/broadcast scheme for DHTs must satisfy three criteria: accuracy, scalability, and robustness. In terms of accuracy, we want the scheme to be able to provide aggregate information with a certain accuracy in a dynamic P2P environment (where nodes frequently join and leave). With respect to scalability, we want to minimize message passing and avoid flooding schemes that generate excessive redundant messages, to make the scheme scalable to a large network. We also want to ensure that there is good load-balancing, in the sense that no node in the network should be responsible for forwarding a disproportionate amount of network traffic. In terms of robustness, a scheme should be resilient to the dynamics of nodes joining, leaving, and failing arbitrarily and independently in a P2P network.

We build and maintain an efficient and robust tree structure in a dynamic P2P environment. We assume that we have at hand an efficient DHT with a circular and continuous namespace, which is true in most existing DHTs. With the underlying DHT infrastructure, we propose a bottom-up approach that constructs and maintains the

*This work is funded in part under NSF Grant ANIR-0137403, "Regions: A new architectural capability in networking", and NSF Grant CCR-0122419, "The Center for Bits and Atoms". It is also funded in part by a gift from the Cisco University Research Program.

tree using soft-state, in which a node dynamically determines its parent using a *parent function*. The particular parent function family we provide allows the efficient construction of multiple interior-node-disjoint trees, thus preventing single points of failure in tree structures, and also distributes the aggregation and broadcast load evenly among nodes. With such an aggregation/broadcast structure, our approach generates a minimal amount of network traffic, and is robust to node failures.

1.1 Organization

The rest of the paper is organized as follows. In Section 2, we first give an overview of DHTs and define parent functions to facilitate our discussion. Then we present a bottom-up tree construction and maintenance algorithm, and analyze aggregate accuracy under node failures. Section 3 discusses desirable properties that a good parent function should have, gives a sample parent function (family), and examines its features. We also look at how parent functions provide a convenient way to construct multiple interior-node-disjoint trees, thus preventing single points of failure. Section 4 discusses implementation issues. Section 5 presents the simulation results on the properties and performance of our scheme. In Section 6, we discuss related work. Section 7 concludes the paper and highlights current areas of ongoing work.

2. A BOTTOM-UP TREE BUILDING ALGORITHM

In this section, we first give an overview on DHTs, and define parent functions. Then we describe bottom-up tree construction and maintenance protocols in detail, and discuss the advantages of such protocols. Finally, we analyze the aggregation/broadcast accuracy in case of node failures.

2.1 Distributed Hash Table Overview

After the success of Napster at file-sharing, many other P2P systems have been proposed, especially DHTs, such as CAN [9], Chord [10], Tapestry [11], or Pastry [12]. DHTs share several common features. First, many DHTs use circular and continuous namespaces, although some are one-dimensional, and some are multiple-dimensional such as CAN. Second, they all provide efficient lookups. Most DHTs can resolve a lookup in $O(\log n)$ or fewer steps, where n is the network size. Finally, they are relatively resilient to node failures in that they automatically repair the network when nodes leave or fail.

In this work, we assume that we have an efficient and robust DHT available as the underlying infrastructure. With these three features (continuous namespace, efficient lookup, and robustness), our goal is to build an efficient and robust aggregation/broadcast tree in a dynamic P2P environment.

2.2 Parent Function Definition

The key idea in our bottom-up tree-building algorithm is to use a many-to-one function, $P(x)$, to map each node uniquely to a parent node in the tree based on its id x. More specifically, the parent node for a node x is the node which owns the id $P(x)$. If node x owns $P^i(x)$ for $i = 1, \cdots, \infty$, then x is the root of the tree. If we consider nodes in a DHT as nodes in a graph and the child-parent relations determined by $P(x)$ to be directed edges, the resulting graph is a directed tree converging at the root.

Definition 1. *A Parent Function, $P(x)$, is a function that satisfies the following conditions:*

$$P(\alpha) = \alpha \quad (1)$$

$$P^{\infty}(x) = \alpha, \ \forall\, x \quad (2)$$

$$Distance(P^{i+1}(x), \alpha) < Distance(P^i(x), \alpha), \ \forall\, i > 0, P^i(x) \neq \alpha \quad (3)$$

where α is an id owned by the root of the tree, x is any valid node id, and $Distance(x, \alpha)$ is the logical distance between id x and the root α, which is essentially the level of id x in the tree.

The above three conditions guarantee that all nodes using $P(x)$ will converge to the root in a finite number of steps, and we prove it in the following theorems.

Theorem 1. *If a function $P(x)$ satisfies the above conditions, there is a directed path from all nodes to the node that owns the id α, which is the root.*

PROOF. Theorem 1 is a direct consequence of *Condition (2)*. Given an arbitrary node x, we know that there is a path from x to $P^i(x), \forall\, i > 0$. Since $P^{\infty}(x) = \alpha$, there must be a path from x to α. □

Theorem 2. *If a function $P(x)$ satisfies the above conditions, all nodes will converge to the root node in a finite number of steps in a finite network.*

PROOF. From *Condition (3)*, we know that a node that owns $P^i(x)$ ($\forall\, i > 0$) is closer to the root α than its child x in terms of $distance$ defined in the parent function. Since there are a finite number of nodes, a node will converge to the root in a finite number of steps, if there is no loop (either back to itself or to a node that is even further away from the parent). We prove that it is impossible to have a loop by contradiction. Without loss of generality, suppose there is a loop in which $P^k(x) = x$ $(k > 0, x \neq \alpha)$. It implies that $Distance(P^k(x), \alpha) = Distance(x, \alpha)$. This contradicts *Condition (3)*. □

Note that the namespace of a DHT is much larger than the normal network size. For example, Chord [10] uses a 160-bit namespace, which can theoretically hold up to 2^{160} nodes. In the widely-deployed P2P network Gnutella, only about $40,000$ peers were observed at a time and $1.2M$ nodes over an 8-day period [13]. Therefore, a node in a DHT is usually responsible for a range of ids, instead of its own id only. Like in Chord [10] and many other DHTs, we assume a node is responsible for the ids or keys between its predecessor (exclusively) and itself (inclusively), and the node is called the id's $successor$ or $owner$. Accordingly, we do not require that a node with an exact id of $P(x)$ exists. Instead, as long as a node is currently responsible for id $P(x)$ according to the underlying DHT, the node represents the id $P(x)$. This rule also applies to the root α such that any node that is currently responsible for id α is the root, but usually the root of a tree is the node that initializes this tree and should be alive through the aggregation/broadcast process.

2.3 Tree Construction Protocol

With a parent function $P(x)$, we can construct an aggregation/broadcast tree by having each node determine its parent node. The tree construction protocol describes how a node joins an existing tree as follows, illustrated in Figure 1.

1. When a new node, say x, joins the P2P network, as most P2P networks assume, node x should already know some node in the network from which it can set up its state. It is also from the introducing node that node x learns about the parent function, $P(x)$, usually parameterized by the root id α.
2. After node x joins the P2P network, it learns the id range that it is responsible for. If α is within this range, node x knows that it has become the new root of the tree. The former root also knows this since node x is its immediate neighbor, and neighbor set maintenance in all current DHTs guarantees that the former root knows the joining of node x. The former root will take actions according to the tree maintenance protocol in Section 2.4.
3. If node x is not the new root, it must find its parent node using $P(x)$. If $P(x)$ falls into its own id range, $P^2(x)$ is computed and checked if it is still in its own id range. This continues until $P^i(x)$ is found not in its range. This is guaranteed to end and a $P^i(x)(i > 0)$ will be found in a finite number of steps by *Theorem 2*.
4. Node x then performs a lookup for the $P^i(x)$. The lookup resulting node, say node y, will become x's parent in the tree.
5. After finding node y, node x sends a message containing $P^i(x)$ to y to register itself as y's child.
6. After receiving x's register message, node y will add node x to its list of child nodes together with the received $P^i(x)$. If node y already has too many children to handle, it will use some admission control to redirect node x to other nodes, as described in the tree maintenance protocol in Section 2.4.

Note that in step 3, a node x can usually find its parent by computing $P(x)$, but due to the sparsity of the P2P network compared with its namespace, it is possible that $P(x)$ is covered in its own id range. In such a case, node x needs to compute $P^i(x)(i > 0)$ where i is the minimum number of times that P has to be applied to x so that $P^i(x)$ maps to a node other than x itself. It can always find its parent in this way unless it is the root. A node can easily figure out whether it is the root by checking if the root id is covered in its id range.

A remaining problem is how to set up the first tree in a DHT, because we need to inform all nodes of the parent function and its parameters first. We choose to build and maintain a default tree with a default parent function $P(x)$ at a default root id α when the P2P network is initialized. We can also depend on either applications or some out-of-band methods to flood the parent function when we want to construct a tree.

2.4 Tree Maintenance Protocol

We need to maintain an aggregation/broadcast tree carefully in the P2P environment where nodes join and leave dynamically, because a single node failure may cause the subtree rooted at the failed node to lose connection with the other parts of the tree. Therefore, it is crucial to maintain a robust tree that can recover quickly from node failures. The key idea is to let each node maintain its link to its parent node independently based on the parent function. Although the parent function gives rise to multiple interior-node-disjoint tree (as will be described in Section 3), here we focus on the robustness of a single tree, since this can be easily extended to a variety of multiple-tree maintenance protocol. The tree maintenance protocol is as follows.

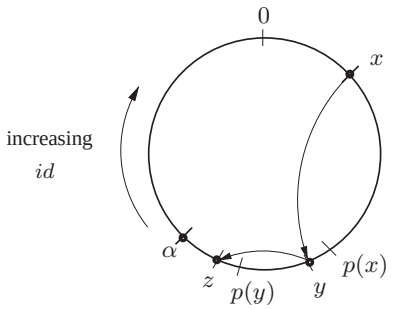

Figure 1: The node joining procedure. The circle represents the namespace, α is the root id, and the arrows show the directed edges of the tree from a child to its parent. In this figure, node y is node x's parent in terms of $P(x)$. Node z is y's parent in terms of $P(y)$.

1. After a node x joins the tree, i.e., registers at its parent node, it is henceforth x's responsibility to contact its parent node y periodically to refresh its status reliably as a child node. The frequency of the refreshment depends on many factors, such as node failure rate and the requirement of applications. If y fails to hear from x after a specified expiry duration, x will be deleted from y's children list.
2. If node y decides to leave the network, it can notify its children and its parent. To do this, y notifies the children and tells them about its successor which should be their new parent. y will also inform its parent, and its parent will simply delete y from its children list. If y leaves without notification, it is considered as node failure.
3. If node y fails, its child x will detect node y's failure when it tries to refresh its status with y. Then x will perform another joining procedure to find its new parent.
4. Node y will discover that it is no longer responsible for the id $P^i(x)$ when a new node, say z, happens to join between $P^i(x)$ and y, and takes over $P^i(x)$. In such a case, y will inform x that it is no longer its parent in terms of $P^i(x)$ and to its best knowledge, z should be its new parent. After receiving y's message, x will contact z. Based on the received $P^i(x)$, z may either add x to its children list, or tell x that to its best knowledge, another node z' should be x's new parent, if it knows that z' is between $P^i(x)$ and z. This recursive procedure continues until a proper new parent is found. Figure 2 shows an example.
5. Node x may notice that it should change the parent. This happens when x's current parent is found in terms of $P^i(x)$ $(\forall\ i > 1)$, which implies that $P^j(x), j = 0, 1, ..., i-1$ are covered in x's id range. If x notices that a new node has joined as its neighbor and is responsible for $P^k(x)(0 < k <$

i), x will switch to the new node and simply stop refreshing its status with its former parent. Figure 3 shows an example.

6. If the current root node finds that a new node has joined as its neighbor and it happens to cover the root id, the old root knows that the new node will take over the root responsibility, so it will find its parent using the parent function.

7. If a parent node is overloaded because it has too many children, or cannot handle all children due to capability changes, it will ask some children to switch to other nodes. The way for the children to find other parents can be based on the parent function, and we will discuss it in detail in Section 3.2.

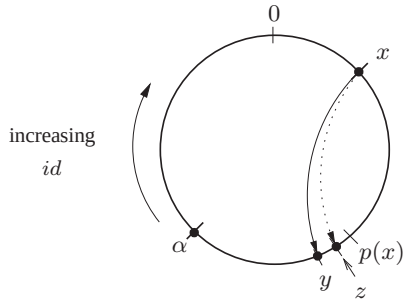

Figure 2: The first case of parent change due to node joining. Node y is node x's current parent in terms of $P(x)$. Node z is a new node which joins and covers $P(x)$. Thus z should be x's new parent. y can easily discover this by observing that z becomes its neighbor.

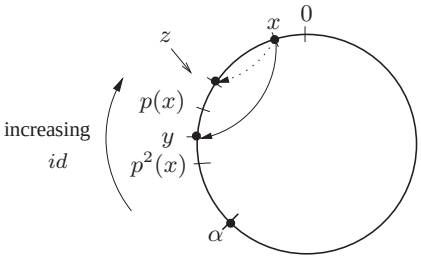

Figure 3: The second case of parent change due to node joining. x's current parent is y, in terms of $P^2(x)$, because there are no nodes between x and $P(x)$. Later, z joins and takes over $P(x)$ and thus should be x's new parent. x can easily discover this by observing that z becomes its neighbor.

2.5 Discussion

As mentioned before, we maintain a default tree on DHTs. We treat it as a light-weight aggregation/broadcast infrastructure, and as a base to construct other special trees, such as trees for media streaming. To keep it simple, we do not consider factors such as bandwidth or proximity in the default tree. Since the refreshing message is small, we can use it to aggregate and broadcast some general network information. For example, we can piggyback the size of the subtree in refreshing messages, so that the root will obtain the size of all its subtrees and learn the network size. Then the root can piggyback the network size in the acknowledgments to its children, so that eventually each node will learn the network size. Other types of aggregate information can also be collected cheaply in similar ways.

There are several advantages of our algorithm over previous tree construction and maintenance schemes. First, our parent-function-based tree is constructed and maintained in a distributed bottom-up way. A parent only needs to passively maintain a list of children without any active detection of their status. Each node is responsible for contacting only one node, i.e., its parent. Therefore, the tree maintenance cost is evenly distributed to each node.

Second, a node's parent is determined by the parent function and node distribution in the namespace, so each node can find its parent independently. Unlike some previous tree-based broadcast and multicast systems where tree repair requires coordination of the root or multiple nodes, our tree can be repaired simultaneously by each node that detects the failure of its parent. Therefore our tree can be repaired easily and efficiently in case of node failure.

Third, parent changes can be detected and completed easily. As explained in Section 2.4, there are two cases when a node should change its parent. Both cases can be detected by a child or a parent by simply observing its neighbor change, as shown in Figure 2 and 3. The first case happens when a new node joins near the parent node. Suppose that a child x registers at its parent y in terms of $P^i(x)$. Later, a new node may join at y's neighborhood and take over $P^i(x)$. It is definitely inefficient to have each node keep detecting whether its corresponding $P^i(x)$ is now covered by another node. Instead, notice that in this case the parent change is triggered by neighbor change *near the parent node*, and neighbor maintenance already exists in most DHTs. Suppose that y is responsible for an id range $(y_0, y]$. If y observes that some nodes have joined between its predecessor y_0 and itself, it checks whether the new nodes are between $P^i(x)$ and y. If so, it knows that its child x should switch to a new node. This detection costs nothing since in all DHTs each node actively maintains its successor and predecessor. y then notifies x about this change. Note that there may be multiple nodes joining the network simultaneously, so y may not know which new node takes over $P^i(x)$ and the responsibility rests on x itself to disambiguate and find out its new parent. The second case happens when a new node joins *near the child*. Suppose x's current parent is found in terms of $P^i(x)(i > 1)$, which implies that $P^j(x), (0 < j < i)$ are covered in x's own range. Later, if x notices that a new node joins at its neighborhood and takes over $P^k(x)(0 < k < i)$, x knows that it should switch to the new node. Therefore, tree maintenance cost on parent changes is very low.

2.6 Accuracy Analysis

A general aggregation operation consists of two steps. First, an aggregation request is broadcast to all nodes from the root. Second, all nodes aggregate data of their own subtrees and forward to their parents. Although tree maintenance protocol tries to recover from node failures and maintain a robust tree, in an extremely transient environment where a significant fraction of the nodes can be down within a short period of time, it is very likely that nodes fail during the broadcast and aggregation procedure. Node failures affect the accuracy in two ways– a node can either fail before forwarding the aggregation request to its children or before forwarding the aggregated result of its subtree to its parent. In both cases, without any recovery mechanism, the final result received by the root will be missing information from the lost subtree. Below we consider the effect of one type of recovery mechanism, namely refreshing, on the conditional probability that the aggregation result is correct, given that aggregation did occur.

We assume that the refreshing/probing is done with respect to a uni-

form distribution with parameter C_r, so the probability distribution function (PDF) is constant at $1/C_r$ from time 0 to time C_r; the lifetime of any node follows an exponential distribution with parameter λ_l; the aggregation time follows a Poisson distribution with parameter λ_a. Let R, L, and A denote the random variables (r.v.) that describe the refreshing time, the lifetime and the aggregation time, respectively. For simplicity, we assume that the aggregation occurs at the expected time, λ_a, and that all the different characteristics of different nodes are independent, unless stated otherwise.

We consider the conditional probability that the aggregation result is correct from a single node's perspective, i.e. its parent has received its aggregation result.

All the probabilities we talk about below are conditioned by the fact that the aggregation occurred at exactly time λ_a. By the total probability theorem, we can split this into different (sub)cases.

1. *Case 1: Parent does not fail within time 0 to time λ_a.* Conditioned further on this, the probability of aggregating correctly for a node is 1. The probability of this case is $\mathbf{P}[L > A | A = \lambda_a] = 1 - \int_0^{\lambda_a} \lambda_l e^{-\lambda_l l} dl = e^{-\lambda_l \lambda_a}$.

2. *Case 2: Parent does fail sometime in between time 0 to time λ_a.* The probability that the parent fails sometime in between time 0 to time λ_a is $\mathbf{P}[L \leq A | A = \lambda_a] = 1 - e^{-\lambda_l \lambda_a}$. There are two subcases to consider, whether the sequence of events is: failure, refresh, aggregation (*Case 2-1*), or refresh, failure, aggregation (*Case 2-2*). In the latter case the result would be incorrect, so we do not need to calculate it. In the former case, we make another simplification: the refresh always has enough time to complete before the aggregation (so refresh is instantaneous). This can be further divided into two worlds.

   (a) *Case A: $\lambda_a \leq C_r$.*

   In essence the probability we are dealing with now is $\mathbf{P}[\text{Aggregation correct for a node} | A = \lambda_a, L \leq A]$
   $= \mathbf{P}[L \leq R | L, R \leq \lambda_a \leq C_r]$
   $= \mathbf{P}[L - R \leq 0 | L, R \leq \lambda_a \leq C_r]$
   (again, L, R denotes the lifetime and the refreshing-time r.v.s respectively). Let W denote the random variable $L - R$. We get the following as the PDF of W after doing the convolution $f_W = \int_l dl f_L(l) f_R(l - w)$:

$$f_W = \begin{cases} 0 & \text{if } w \geq \lambda_a \\ \int_{l=0}^{\lambda_a - w} \frac{1}{\lambda_a} \lambda_l e^{-\lambda_l l} dl & \text{if } 0 \leq w \leq \lambda_a \\ \int_{l=-w}^{\lambda_a} \frac{1}{\lambda_a} \lambda_l e^{-\lambda_l l} dl & \text{if } -\lambda_a \leq w \leq 0 \\ 0 & \text{if } w \leq -\lambda_a \end{cases}$$

   This evaluates to

$$f_W = \begin{cases} 0 & \text{if } w \geq \lambda_a \\ \frac{1}{\lambda_a}[1 - e^{-\lambda_l(\lambda_a - w)}] & \text{if } 0 \leq w \leq \lambda_a \\ \frac{1}{\lambda_a}[e^{\lambda_l w} - e^{-\lambda_l \lambda_a}] & \text{if } -\lambda_a \leq w \leq 0 \\ 0 & \text{if } w \leq -\lambda_a \end{cases}$$

   Therefore,

$$\begin{aligned} &\mathbf{P}[L - R \leq 0 | L, R \leq \lambda_a \leq C_r] \\ &= \int_{w=-\lambda_a}^{0} \frac{1}{\lambda_a}[e^{\lambda_l w} - e^{-\lambda_l \lambda_a}] \\ &= \frac{1}{\lambda_a}[1/\lambda_l - (1/\lambda_l + \lambda_a) e^{-\lambda_l \lambda_a}]. \end{aligned}$$

   (b) *Case B: $\lambda_a > C_r$.*

   Here, we are after the same probability $\mathbf{P}[\text{Aggregation correct for a node} | A = \lambda_a, L \leq A]$, which in this case simplifies to $\mathbf{P}[L - R \leq 0 | L, R \leq \lambda_a, \lambda_a \geq C_r]$. Again, we carry out the convolution, and get:

$$\begin{aligned} &\mathbf{P}[L - R \leq 0 | L, R \leq \lambda_a, \lambda_a \geq C_r] \\ &= \frac{1}{C_r}[1/\lambda_l - (1/\lambda_l) e^{-\lambda_l C_r} - C_r e^{-\lambda_l \lambda_a}]. \end{aligned}$$

We can now calculate the probability of the aggregation being correct from a single node's perspective, applying independence:

$$\begin{aligned} &\mathbf{P}[\text{Aggregation. is correct for a node} | \text{Aggregation. occurred at } \lambda_a] \\ &:= \mathbf{P}[\text{correct} | A = \lambda_a] \\ &= \mathbf{P}[\text{correct} | A = \lambda_a, L > A] \cdot \mathbf{P}[L > A | A = \lambda_a] \\ &\quad + \mathbf{P}[\text{correct} | A = \lambda_a, L \leq A] \cdot \mathbf{P}[L \leq A | A = \lambda_a] \\ &= 1 \cdot e^{-\lambda_l \lambda_a} \\ &\quad + \mathbf{P}[\text{correct} | A = \lambda_a, L \leq A] \cdot (1 - e^{-\lambda_l \lambda_a}) \end{aligned}$$

At this point, depending on whether we are in case A ($\lambda_a \leq C_r$) or case B ($\lambda_a > C_r$), the result is going to be different. Recall that we are fixing $A = \lambda_a$, so we are either in case A or case B, and there is no probability distribution over it.

In case A ($\lambda_a \leq C_r$) we have:

$$\begin{aligned} &\mathbf{P}_A := \mathbf{P}[\text{correct} | A = \lambda_a] \\ &= 1 \cdot e^{-\lambda_l \lambda_a} \\ &\quad + 1 \cdot \frac{1}{\lambda_a}[1/\lambda_l - (1/\lambda_l + \lambda_a) e^{-\lambda_l \lambda_a}] \\ &= \frac{1}{\lambda_a \lambda_l}[1 - e^{-\lambda_a \lambda_l}], \end{aligned}$$

and in case B ($\lambda_a > C_r$) we have:

$$\mathbf{P}_B := \mathbf{P}[\text{correct} | A = \lambda_a] = \frac{1}{C_r \lambda_l}[1 - e^{-C_r \lambda_l}].$$

For instance, when C_r = 10, λ_a = 20, and λ_l = 0.1, then according to $\mathbf{P}_B$, the probability of aggregation being correct from a single node's perspective is 63.2%. More examples are summarized in Table 1.

3. PARENT FUNCTION

Protocols in Section 2 help to construct and maintain a tree in a P2P network, but they do not determine the shape of a tree and other

Setting			Probability	
C_r	λ_a	λ_l	$\mathbf{P}_A$	$\mathbf{P}_B$
20	10	0.001	99.5%	N/A
20	10	0.005	97.5%	N/A
20	10	0.01	95.2%	N/A
20	10	0.02	90.6%	N/A
20	10	0.03	86.4%	N/A
20	10	0.04	82.4%	N/A
10	20	0.05	N/A	78.7%
10	20	0.06	N/A	75.2%
10	20	0.07	N/A	71.9%
10	20	0.08	N/A	68.8%
10	20	0.09	N/A	65.9%
10	20	0.1	N/A	63.2%

Table 1: The probability of aggregation being correct from a single node's perspective under different settings. Note that λ_l is the inverse of the average node life time.

properties: those properties are determined by the parent function. Parent functions play a central role in the properties of an aggregation/broadcast tree. In this section, we discuss the desirable features that a parent function should have. Then we present a sample parent function family, show that it satisfies our definition (i.e. satisfies the three required conditions mentioned in Section 2.2), and analyze its properties.

3.1 Desirable Features

Parent function determines the properties of the constructed tree. Therefore, we believe that, due to the scale and dynamics of DHTs, a good parent function should have the following features to be efficient and flexible, besides the three required conditions mentioned in Section 2.2.

First, for the purpose of load balancing, it should guarantee that each parent node has approximately the same number of children, given that nodes are uniformly distributed in the namespace. This helps to build a balanced tree.

Second, it should guarantee that node joining and leaving will not greatly affect the structure of an established tree. We can identify that two features of a P2P network make it hard to stabilize a tree. One is that, it may sound appealing to have nodes at each level of a tree be evenly distributed in the namespace. However, a DHT network size is much smaller than its namespace, so it is very likely that an id of $P(x)$ will map to the node that follows $P(x)$. A well-intended parent function that makes nodes at each level of a tree evenly distributed will inevitably lead to chaos in a real tree due to the sparsity of the namespace. Therefore, a good parent function should guarantee an approximately balanced tree structure under this circumstance. The other feature that makes maintaining a stable tree difficult is that a node may change its parent node due to nodes joining and leaving. For example, suppose a node x with a large subtree attaches to a parent node at a high level y. Then a new node z joins and x should switch from y to z according to the parent function. If z is a leaf node in a low level, then the resulting tree is very unbalanced.

Third, a parent function should have good support for branch balancing in a dynamic environment. Although statistically a good parent function can balance the number of children each node has, it is unavoidable that some nodes may be assigned too many children to handle due to the dynamics of P2P networks and variance in node distribution. There are two aspects of branch balancing: admission control and dynamic adaptation. In admission control, a parent decides whether to accept a child when the child tries to register. Admission control is not enough since a parent's condition may change such that it can no longer handle all children, and in such a case, a dynamic adaptation is needed. In both cases, the key problem is how the refused or abandoned children find their new parents. If a node's current parent is overloaded and it has multiple parent candidates, it can switch to another parent.

3.2 A Sample Parent Function

There are many functions that satisfy Conditions (1) to (3) in Section 2.2. The following is an example adopted in this work[1]:

$$P_s(x) = \begin{cases} \alpha + \left\lfloor \frac{(x-\alpha) \pmod m}{k} \right\rfloor \pmod m, \\ \quad \text{for } 0 \le (x-\alpha) \pmod m < \frac{m}{2} \\ \alpha - \left\lfloor \frac{m-(x-\alpha) \pmod m}{k} \right\rfloor \pmod m, \\ \quad \text{for } \frac{m}{2} \le (x-\alpha) \pmod m < m \end{cases}$$

where x is the id of the present node, α is the root id, k is a parameter that determines the branching factor of a tree, and $m = 2^s$, where s is the number of bits for the address space.

As shown in Figure 4, a tree resulting from this function is rooted at a node that owns the id α. The expected branching factor of a spanning tree constructed with this function is approximately k if the nodes are uniformly distributed in the namespace (except for the root). The expected height is $O(\log_k n)$, where n is the size of the network. Before proving these properties in theorems 4 and 5 respectively, we first show that $P_s(x)$ is indeed a parent function, as per definition 1.

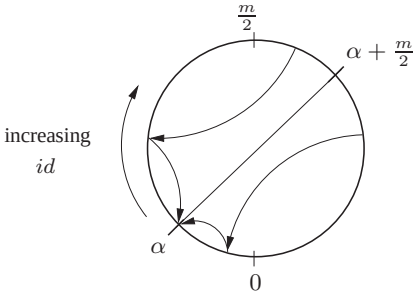

Figure 4: Aggregation pattern of the sample parent function. The circle represents the namespace, α is the root id, m is the size of the namespace, and the arrows show the directed edges of the tree from a child to its parent.

Theorem 3. *$P_s(x)$ is a parent function.*

PROOF. This follows from lemmas 1, 2, and 3. □

[1]Note that the modulo operation on negative numbers in $P_s(x)$ is defined as follows: $a \bmod b = b - (-a) \bmod b$, if $-b < a < 0$.

Lemma 1. *$P_s(x)$ satisfies the first condition of definition 1, i.e. $P_s(\alpha) = \alpha$.*

PROOF. Direct substitution yields the desired result. □

Lemma 2. *$P_s(x)$ satisfies the second condition of definition 1, i.e. $P_s^{\infty}(x) = \alpha,\ \forall\ x$.*

PROOF. The essence of this parent function is that after each iteration, the distance between the root and the current id $P_s(x)$ is reduced by branching factor k. To make it simple, we make the following substitution.

Let d_i be the distance between the current id and the root α after i iterations. We can convert the parent function as follows.

1. When $0 \leq (x-\alpha) \pmod m < \frac{m}{2}$, we have:

$$\begin{aligned} d_0 &= (x-\alpha) \pmod m; \\ d_{i+1} &= \frac{d_i}{k}; \\ \Rightarrow P_s^i(x) &= (\alpha + d_i) \pmod m; \end{aligned}$$

   It is easy to see that $d_i = \frac{d_0}{k^i}$. When $i \rightarrow \infty, d_i \rightarrow 0$, and thus $P_s^i(x) \rightarrow \alpha$.

2. Similarly, when $\frac{m}{2} \leq (x-\alpha) \pmod m < m$, we have:

$$\begin{aligned} d_0 &= (m - (x-\alpha)) \pmod m; \\ d_{i+1} &= \frac{d_i}{k}; \\ \Rightarrow P_s^i(x) &= (\alpha - d_i) \pmod m; \end{aligned}$$

   It is easy to see that $d_i = \frac{d_0}{k^i}$. When $i \rightarrow \infty, d_i \rightarrow 0$, and thus $P_s^i(x) \rightarrow \alpha$.

Thus $P_s^i(x) \rightarrow \alpha$ in both cases. □

Lemma 3. *$P_s(x)$ satisfies the third condition of definition 1, i.e. $Distance(P_s^{i+1}(x), \alpha) < Distance(P_s^i(x), \alpha),\ \forall\ i > 0$,*

where $Distance(x, \alpha) =$

$$\begin{cases} (x-\alpha)\ (\textit{mod}\ m), & \textit{for}\ 0 \leq (x-\alpha)\ (\textit{mod}\ m) < \frac{m}{2} \\ m - (x-\alpha)\ (\textit{mod}\ m), & \textit{for}\ \frac{m}{2} \leq (x-\alpha)\ (\textit{mod}\ m) < m \end{cases}$$

PROOF. In the proof of property 2, it is easy to see that $d_{i+1} < d_i$ in both cases, and therefore we have

$Distance(P_s^{i+1}(x), \alpha) < Distance(P_s^i(x), \alpha),\ \forall\ i > 0$. □

Therefore $P_s(x)$ is indeed a parent function. We now move on to discuss the expected branching factor and the expected height of a tree constructed using $P_s(x)$.

Theorem 4. *If the node distribution in the namespace is uniform, then k is the expected branching factor of the tree constructed with $P_s(x)$ (with the exception being the root who has $2k-2$ children).*

PROOF. Suppose there are a total of n nodes in the namespace with namespace size m. Since nodes are evenly distributed in the namespace, on average each node takes charge of $\frac{m}{n}$ fraction of the id space, i.e. node x will take charge of $(x - \frac{m}{n}, x]$, and together with its following $k-1$ successors, they cover an id range of $(x - \frac{m}{n}, x + (k-1)\frac{m}{n}]$. Without loss of generality, suppose that for any id, y, in this range it satisfies the property that $0 \leq (y-\alpha) \pmod m < \frac{m}{2}$. Then, based on the parent function, the parent(s) of this combined id range will be $(\alpha + \frac{x - \frac{m}{n} - \alpha}{k}, \alpha + \frac{x + (k+1)\frac{m}{n} - \alpha}{k}]$, which covers exactly $\frac{m}{n}$ fraction of the id space, the same as the average id range of a node. Therefore, the k nodes within $(x - \frac{m}{n}, x + (k-1)\frac{m}{n}]$ will map to a single parent. In the case of the root node, since nodes converge both in the clockwise and counterclockwise directions (recall we assume that namespaces are circular), it has $2k-2$ children. Therefore, we conclude that given a uniform node distribution in the namespace, each non-leaf node will have k children whereas the root will have $2k-2$ children with high probability. □

Theorem 5. *If the node distribution in the namespace is uniform, then $O(\log_k n)$ is the expected height of a tree constructed with $P_s(x)$, where n is the size of the network.*

PROOF. With the previous property, we know that each non-leaf node has about k children except that the root has $2k-2$ children. Suppose that there are a total of n nodes. For a tree with height h, it must satisfy the following two inequalities:

$$\begin{aligned} n &\leq 1 + 2k - 2 + 2(k-1)k + \cdots + 2(k-1)k^{h-1} \\ &= 1 + 2(k^h - 1) \\ \Rightarrow h &\geq \log_k \frac{n+1}{2} \end{aligned}$$

$$\begin{aligned} n &> 1 + 2k - 2 + 2(k-1)k + \cdots + 2(k-1)k^{h-2} \\ &= 1 + 2(k^{h-1} - 1) \\ \Rightarrow h &< \log_k \frac{n+1}{2} + 1 \end{aligned}$$

Combining the two inequalities, we get $h = \lceil \log_k \frac{n+1}{2} \rceil$. Therefore, given that nodes are uniformly distributed in the namespace, the height of a tree with size n is $O(\log_k n)$. □

With the above theorems, we now briefly discuss how our sample parent function $P_s(x)$ satisfies the three desirable features we discussed in Section 3.1.

First, our parent function builds a balanced tree. According to Theorem 4, each non-leaf node has on average k children except the root. The root has $2k-2$ children because children from both sides converge to it, as shown in Figure 4 (Node x for which $(x-\alpha) \pmod m \in [0, \frac{m}{2})$ converges counterclockwise to the root, while node x for which $(x-\alpha) \pmod m \in [\frac{m}{2}, m)$ converges clockwise.).

Second, the tree constructed from our parent function is resilient to node joining and leaving, because, according to the proof of Theorem 4, neighboring nodes are at the same level or the adjacent levels in the tree, and such parent changes will usually move children from a node to another node at the same or an adjacent level of the tree. On the other hand, if a parent function maps nodes' parents evenly in the namespace, its tree will tend to be unbalanced.

Third, our parent function does not provide natural support for branch balancing, but can be easily enhanced as follows. In case of overloading, the parent will ask some of the farthest children in the namespace to move. An alternative parent candidate will be found by moving stepwise one neighbor of the parent away from the root, repeatedly until an underloaded node is found. This is guaranteed to terminate because a leaf node will eventually be found in the worst case. After a certain time, the moved children will recompute the parent function, and move back to their normal parents. A nice property of our parent function for branch balancing is that this has little impact on the height of the tree, because nodes near each other are in the same level or adjacent levels of the tree and thus those temporary parents are probably in the same level as the original parent. As a result, the convergence time will not be affected significantly, and the root will not receive redundant information from different aggregation paths.

3.3 Overcoming Single Points of Failure

A common problem of using trees aggregation and broadcast is that trees inherently present single points of failure. The bottom-up approach will experience periodic glitches in a highly dynamic network.

To address this problem and improve the robustness, we can build multiple trees, and compute the same aggregation/broadcast over them. By distributing the ids of these trees over the namespace uniformly and adopting some quorum system or by averaging over the estimates obtained from several trees, we can further improve the robustness of the aggregation/broadcast.

Parent functions provide a convenient way to build multiple interior-node-disjoint trees. The key is to adjust parameters in a parent function, so that we can get a family of similar parent functions. Trees constructed using that family of parent functions do not overlap in their interior nodes. Here we show that our sample parent function can easily be used to construct disjoint trees as follows. If a tree is rooted at id 0, then according to the parent function, its non-leaf nodes will be mostly in [0, $\frac{m}{2 \cdot k}$] and [$m - \frac{m}{2 \cdot k}$, $m - 1$], which is a range of length $\frac{m}{k}$ in the namespace. Nodes in the other areas are all leaf nodes. Generally, under this parent function, non-leaf nodes in a tree rooted at α are mostly in $[\alpha - \frac{m}{2 \cdot k} (\text{mod } m), \alpha + \frac{m}{2 \cdot k} (\text{mod } m)]$, and all other nodes are leaf nodes. Therefore, if we choose k similar parent functions and the distance between two neighboring roots is $\frac{m}{k}$, we can construct k interior-node-disjoint trees, because paths from any two roots to a node in the two trees are disjoint. In this way, we can greatly increase the tolerance of node failures and guarantee that with high probability every node will receive the message at least once.

4. PRACTICAL IMPLEMENTATION

The construction and maintenance protocols and the parent function help us build a robust tree as a default aggregation/broadcast mechanism. In practice there are several other issues, which are addressed in this section.

4.1 Two Operation Modes

As we mentioned before, the tree maintenance messages, i.e., the refreshment and acknowledgment messages, can be used for some light-weighted information aggregation and broadcast. For example, we can compute network size in this way, since it adds little to bandwidth consumption. This is an aggregation/broadcast in the background, which we call the *default mode*.

However, a node may need to perform a special aggregation and/or broadcast that is useful only for the node itself, for example, searching for files whose names contain certain key words. We call this the *on-demand mode*. The on-demand mode of aggregation consists of the following:

1. When a node wants to perform an aggregation using the tree, it sends a request to the root.
2. Upon receiving a request, the root broadcasts it to its immediate children in the tree, which in turn forward the request to their children.
3. When a leaf node receives the request, it performs the corresponding aggregate operation, and sends back the result with a timestamp to its parent. If the child does not receive an acknowledgment from its parent within a certain time interval, it determines its new parent and sends the results. Note that it takes some time to find the new parent.
4. A parent node waits for results from all its children. If it does not hear from a child after an expiry time, it will delete this child and not wait for its data. After receiving data from all children, the parent node performs the aggregate operation, attaches a timestamp, and forwards the aggregate result to its parent.
5. If a node receives data from a new child after having sent its aggregate result, it will compute and forward the new result to its parent.
6. If a node receives data from a child several times, but with different timestamps, it will compute the latest result and forward it to its parent.
7. After the root receives results from all children, it will wait for an additional amount of time that is long enough to allow information delayed by parent failures to reach the root. In our implementation, the time equals the height of the tree times the average round trip time. Then the root processes all collected data and computes the final aggregation result. Finally, it sends the result to the request node.

4.2 Constructing Additional Trees

In a practical system, we may want the capability to set up trees rooted at arbitrary node ids to distribute the load of a root and to provide robustness. Depending on specific applications, we can either rely on the application to disseminate or embed the root identifier, or have one permanent tree rooted at a pre-determined well-known id α_0, as described in Section 2.3. We focus on the second case. In this case, if a node, say x, wants to construct a new tree rooted at itself, it will send a message to the root of the default tree, specifying the parent function with the root id as its own id. The message will then be broadcast down the default tree. All nodes will eventually receive this message, and participate in the new tree rooted at x.

When a new node joins the network, it will get a list of all existing trees from its parent node. Nodes also periodically exchange information on the existing trees. In this way, all nodes will eventually discover and participate in all existing trees.

When the root node of a tree wishes to tear down its tree, it will simply broadcast a message to tell all nodes that the tree should be torn down. If a node does not receive the tear-down message due to its parent failure, it will still get it when it detects the parent failure and switches to a new parent which has already received the message. Therefore, the message is guaranteed to reach all nodes eventually. If a root fails without notification, subsequent messages will end up at a succeeding node which discovers that it has become the root of a tree that it did not set up. It can decide either to keep the tree or to broadcast a message to tear it down.

5. PERFORMANCE EVALUATION

In this section, we evaluate the performance of our bottom-up tree-building algorithm using the parent function in Section 3.2. As an example of usage, we evaluate the estimation of available network storage under the default mode, and the estimation of network size under the on-demand mode.

5.1 Simulation Setup

Our experiments are divided into discrete time periods. In each period, a number of nodes join the network according to a Poisson distribution with parameter λ_j. At the point when a node joins, its departure time is set according to an exponential distribution with parameter λ_l, and nodes are removed from the network when their lifespans expire. In the default mode, each node sends a refreshment message that contains the aggregate information to its parent in each period. In the on-demand mode, a node refreshes its status under a uniform distribution with parameter C_r, and the aggregation time follows a Poisson distribution with parameter λ_a. In all experiments, we use the parent function in Section 3.2. The root id α is set to 0, and k is 4.

Our simulations were written in Java. The experiments were run on single-processor Intel P4-2.2GHz machine with 1GByte RAM running Linux with Java ™ 2 Runtime Environment version 1.4.2.

5.2 Tree Properties

In the first experiment, we evaluated the overhead of our tree construction and maintenance algorithm in terms of network traffic by counting the number of messages sent in the tree construction process. In this experiment, the node failure rate is approximately λ_l = 10% per time period, and about 10% new nodes join the network too. This keeps the network size roughly stable. During each period, each node refreshes its status with its parent once. The communication cost in terms of messages sent as a function of network size is shown in Figure 5.

The total number of messages in each period is at least twice the network size, because each node sends a refreshment message to its parent and receives an acknowledgment (except for the root). Note that refreshment messages can also be used to aggregate and broadcast information in the default mode. Additional messages are needed for new nodes to join, to repair the tree in case of nodes failures, and for overloaded nodes to do branch-balancing. Figure 5 shows that the number of additional overhead messages (including tree-repair messages, branch-balancing messages, and node-joining messages) is small, compared with the total number of messages.

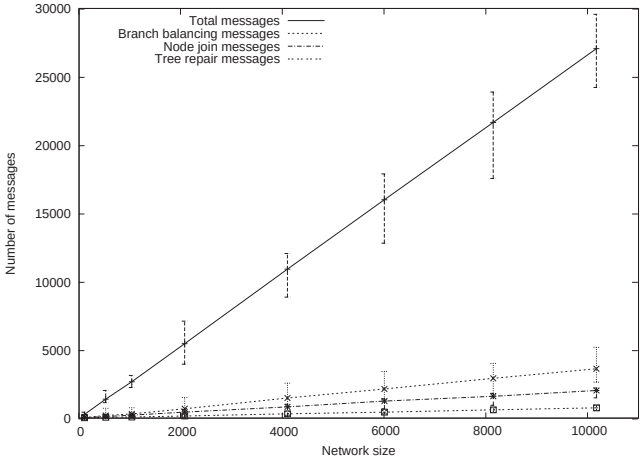

Figure 5: Communication overhead for tree construction.

An important property is the number of branches or children at non-leaf nodes, as it reflects the overhead of a parent node. Ideally, we would like all the intermediate nodes to have approximately the same number of children.

In the simulation, we define an overloaded node as having more than $2k$, i.e. 8 children. We use the branch balancing algorithm discussed in Section 3.2, which moves additional children to its neighbor towards the leaf side. We find that, without branch balancing, there are 3% of overloaded nodes in each period, and the overloading lasts about 3 periods on average. Overloading is automatically resolved when children leave or new nodes join to take over some children from the current parent. The branch balancing procedure usually propagates within 2 neighbors. Figures 6 shows the relationship between network size and branch. We can see that without branch balancing, an overloaded node can take as many as 48 children in a network with about 12800 nodes, and with branch balancing, the maximum branch is only 8, which is the upbound of the number of children a node can have.

The height of the resulting tree affects the performance since it determines the time it takes for the information to propagate from leaf nodes to the root. Figure 7 shows the average tree heights with or without branch balancing under different network sizes. We can see that our branch balancing scheme does not increase the height of the tree, even though the branch balancing affects the tree structure since it does not obey the parent function.

5.3 Network Storage Estimation

As an example of usage, we use our bottom-up tree to estimate the evolution of available network storage in the default mode, which aggregates continuously. We estimate network size in a similar way.

Figure 8 shows the evolution of the network storage estimate and the network size estimate at the root of the tree with a node failure rate of λ_l = 5% per time period. The storage on each node keeps changing according to an approximately Gaussian distribution with a mean of 50MB and a standard deviation of 20MB. The simulation consists of three stages: increasing, stable, and decreasing. From Figure 8, we can see that the average estimation is very close to the true value. Specifically, in the increasing stage, the estimates tend

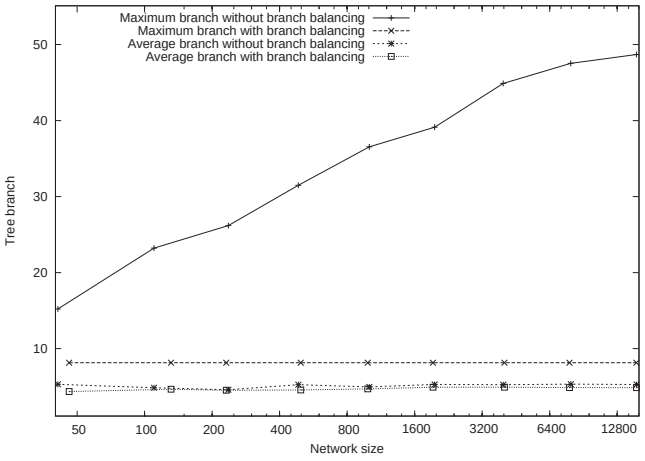

Figure 6: Tree branches against network size.

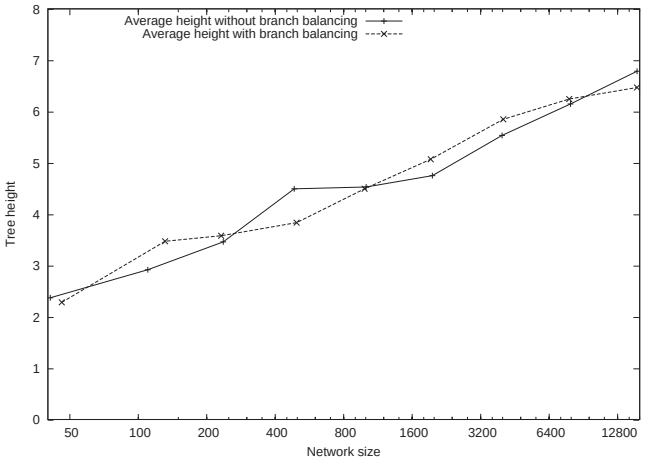

Figure 7: Tree heights against network size.

to be smaller than the actual network storage because there is a lag between the estimate and the actual value. Similarly, during the decreasing stage, the estimates are usually higher than the actual storage size. The spikes are caused by the failure of intermediate nodes high up in the tree, leading to temporary losses or duplicates in the storage information. The results demonstrate that our algorithm is responsive to network storage changes, and recovers rapidly from such failures.

5.4 On Demand Estimation

We also evaluate the accuracy of on-demand estimation in the simulation. In this setting, λ_l = 0.05, λ_a = 50, C_r = 10. Based on the analysis in Section 2.6, we know that with 78.7% probability, a node can send the aggregate result successfully and timely even in case of parent node failure. The average estimation over 20 trials is plotted. We set up an initial network of 10000 nodes. During the aggregation, a certain fraction of nodes fail. Figure 9 shows the simulation results under various node failure rates. *Before Aggregation* refers to the actual network size before aggregation, and *After Aggregation* refers to the actual network size after aggregation. Since it takes some time for an aggregation procedure to complete, we consider that an estimated value is correct if it is between the

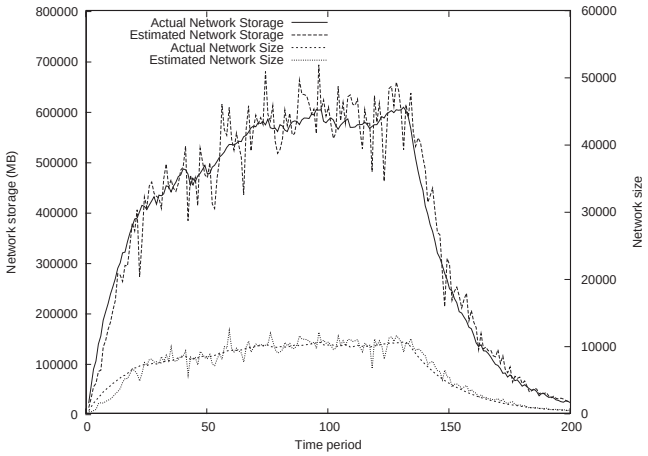

Figure 8: Network storage and network size estimation. The solid curve at the top shows the evolution of the actual network storage, and the dashed curve with spikes at the top shows the estimation of network storage. The dashed line at the bottom shows the evolution of network size, and the dotted line with spikes at the bottom shows the estimation of network size.

original network size and the network size at the end of aggregation. From Figure 9 we can see that in most cases the estimation is within the correct range. When node failure rate reaches 15%, some estimates start falling out of the range.

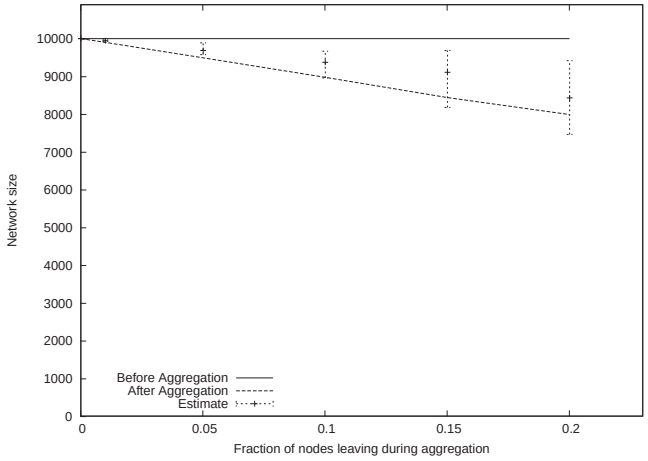

Figure 9: On demand estimation under various node failure rates.

6. RELATED WORK

There is a large body of literature in the area of broadcast. In general, existing schemes can be categorized as flooding-based approaches ([14, 15]) and top-down spanning-tree approaches ([16, 17]). A major drawback of the former is to generate redundant traffic, while a major drawback of the latter is that when nodes fail in the middle of the hierarchy, large sections of the original spanning tree will lose contact with the root, and reconstruction of the subtrees requires significant efforts. In contrast, we try to solve the problem under a more dynamic environment, and our bottom-up approach not only enables fast recovery from such failures, but also provides an easy way to build multiple trees with disjoint paths

so that all nodes can be reached with high probability.

Most of previous P2P multicast systems ([18, 19, 20, 16, 21]) are tightly coupled with underlying P2P networks. For example, in [16], El-Ansary et al. discussed how to use routing tables to construct a broadcast tree, which can be viewed as a special case of our scheme. The advantage of using routing table information is that the child maintenance cost is saved by retrieving children from the routing table, which is always kept updated by the underlying P2P network. The disadvantage is that the tree structures are constrained by the underlying P2P topologies and thus not flexible. Compared with these systems in which it is hard for a node to adjust the number of its children based on its capability, our bottom-up tree scheme is independent of underlying overlay and its structure is determined by the parent function, which is flexible in building trees according to different requirements.

In [22], Bawa et al. proposed top-down flooding-based aggregation schemes over unstructured P2P networks, which require a known universal maximum delay and a known upper bound to the network diameter. With the progress in structured P2P networks, we believe that it is feasible to build and maintain a robust DHT infrastructure at relatively low cost without those assumptions, and it is useful to attempt to solve the aggregation problem over DHTs using a more efficient and scalable approach.

Our bottom-up approach of using a function to map nodes onto parent nodes is similar in spirit to the generalized scheme for building DHTs proposed by Naor and Wieder [23]. In their scheme, functional mapping is used to build the general DHT infrastructure, while in our scheme we seek to build a specific tree structure over underlying DHT infrastructure for aggregation and broadcast. In [24], Zhang et al. built a data overlay using a top-down method, while again, our scheme focuses on the aggregation/broadcast problem and is bottom-up.

In CoopNet ([25]), the root node coordinates all tree management functions, including construction and maintenance. This centralized management makes tree maintenance easy. Trees are constructed incrementally, so the position of a node in the tree depends on when it joins the tree. In our approach, each node can find its position in the tree without asking any central point, and its position is fixed by the parent function. We do not consider bandwidth, since our intention is not for stream media, but for a general lightweight aggregation infrastructure, whose bandwidth requirement is very low.

With respect to dealing with single points of failure, our approach is similar to Splitstream ([26]) in building multiple interior-node-disjoint trees. Splitstream builds multiple interior-node-disjoint trees over Pastry. Like most previous work on multicast over P2P networks, trees are built using the routing table entries, and thus the properties and the robustness of the multicast tree relies on the underlying DHT. In our approach, the tree is constructed based on parent functions. Therefore, different applications can build trees based on their requirements using different parent functions. Another advantage of our tree construction is that each node maintains its own parent link and the tree can be repaired simultaneously. In Splitstream, multiple disjoint trees are built based on the base of the id space, while our approach is independent of any underlying routing topology, and thus makes it easier to construct and control the number of trees by simply changing the root id and the branch factor.

7. CONCLUSIONS AND FUTURE WORK

The aggregation/broadcast problem for P2P networks is complicated by the scale and dynamics of such networks. We propose to construct a robust tree over the P2P network. The reason for separating the solution to aggregation/broadcast from the underlying network is that we concentrate on a general-purposed capability independent of, but needed in, different P2P systems. We have presented a bottom-up approach by mapping from a continuous function into the discrete id space. The major advantage is that it has a relatively low overhead and is resilient to node failures. Our scheme is also flexible in that parameters in the parent function can be used to control the tree structure and characteristics such as the height and the branch factor. We also presented the notion of parent function families to aid in building multiple interior-node-disjoint trees, which helps to solve the problem of single points of failure.

Parent functions play an important role in our bottom-up approach. We have only thus far discussed several properties of a parent function that help to improve the tree performance. By adopting a different parent function, we may be able to improve some aspects of the system, or at least tradeoff between various costs and benefits, which makes our results more widely applicable since one could conceivably define the cost and benefit function for a particular application and then find a parent function that maximizes the benefit/cost ratio.

8. ACKNOWLEDGMENTS

We would like to thank Ben Leong and members of ANA group at MIT CSAIL for helpful discussions. We also thank John Wroclawski, Klara Nahrstedt, and the anonymous reviewers for their insightful feedback and comments on the initial draft of this paper.

9. REFERENCES

[1] Napster, “http://www.napster.com,” .

[2] Gnutella, “http://gnutella.wego.com,” .

[3] KaZaA, “http://www.kazaa.com,” .

[4] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong, “Freenet: A distributed anonymous information storage and retrieval system,” *Lecture Notes in Computer Science*, vol. 2009, pp. 46+, 2001.

[5] Athicha Muthitacharoen, Robert Morris, Thomer M. Gil, and Benjie Chen, “Ivy: A read/write peer-to-peer file system,” in *Proceedings of 5th Symposium on Operating Systems Design and Implementation*, 2002.

[6] Ben Paechter, T. Baeck, M. Schoenauer, A.E. Eiben, and J. Merelo, “A distributed resource evolutionary algorithm machine,” in *Proceedings of the 2000 Congress on Evolutionary Computation (CEC 2000)*, San Diego, USA, 2000, IEEE, pp. 951–958, IEEE Press.

[7] Dahlia Malkhi, Moni Naor, and David Ratajczak, “Viceroy: A scalable and dynamic emulation of the butterfly,” in *Proceedings of 21st ACM Conf. on Principles of Distributed Computing (PODC’02)*, July 2002.

[8] Gurmeet Manku, Mayank Bawa, and Prabhakar Raghavan, “Symphony: Distributed hashing in a small world,” in *Proceedings of 4th USENIX Symposium on Internet Technologies and Systems*, March 2003.

[9] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker, "A scalable content addressable network," in *Proceedings of ACM SIGCOMM 2001*, 2001.

[10] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan, "Chord: A scalable Peer-To-Peer lookup service for internet applications," in *Proceedings of the 2001 ACM SIGCOMM Conference*, 2001, pp. 149–160.

[11] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph, "Tapestry: An infrastructure for fault-tolerant wide-area location and routing," Tech. Rep. UCB/CSD-01-1141, UC Berkeley, April 2001.

[12] Antony Rowstron and Peter Druschel, "Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems," in *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, Nov. 2001, pp. 329–350.

[13] Stefan Saroiu, P. Krishna Gummadi, and Steven D. Gribble, "A measurement study of peer-to-peer file sharing systems," in *Proceedings of Multimedia Computing and Networking 2002 (MMCN '02)*, San Jose, CA, USA, January 2002.

[14] M.-J. Lin, K.Marzullo, and S.Masini, "Gossip versus deterministic flooding: Low message overhead and high reliability for broadcasting on small networks," Tech. Rep. Technical Report CS1999-0637, University of California, San Diego, 1999.

[15] M. Portmann and A. Seneviratne, "Cost-effective broadcast for fully decentralized peer-to-peer networks," *Computer Communication*, vol. Special Issue on Ubiquitous Computing, Elsevier Science, 2002.

[16] Sameh El-Ansary, Luc Onana Alima, Per Brand, and Seif Haridi, "Efficient broadcast in structured P2P networks," in *2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkeley, CA, 2003.

[17] Y. Chawathe and M. Seshadri, "Broadcast federation: An application-layer broadcast internetwork," in *Intnl. Workshop on Network and Operating Systems Support for Digital Audio and Video*, 2002.

[18] Antony I. T. Rowstron, Anne-Marie Kermarrec, Miguel Castro, and Peter Druschel, "SCRIBE: The design of a large-scale event notification infrastructure," in *Networked Group Communication*, 2001, pp. 30–43.

[19] Shelley Q. Zhuang, Ben Y. Zhao, Anthony D. Joseph, Randy H. Katz, and John D. Kubiatowicz, "Bayeux: an architecture for scalable and fault-tolerant wide-area data dissemination," in *Proceedings of the 11th international workshop on Network and operating systems support for digital audio and video*. 2001, pp. 11–20, ACM Press.

[20] Sylvia Ratnasamy, Mark Handley, Richard Karp, and Scott Shenker, "Application-level multicast using content-addressable networks," in *3rd International Workshop on Networked Group Communication*, November 2001.

[21] Rongmei Zhang and Y. Charlie Hu, "Borg: A hybrid protocol for scalable application-level multicast in peer-to-peer networks," in *The 13th International Workshop on Network and Operating Systems Support for Digital Audio and Video (NOSSDAV)*, Monterey, California, 2003.

[22] M. Bawa, H. Garcia-Molina, A. Gionis, and R. Motwani, "Estimating aggregates on a peer-to-peer network," in *Submitted for publication*, 2003.

[23] Moni Naor and Udi Wieder, "Novel architectures for P2P applications: the continuous-discrete approach," www.wisdom.weizmann.ac.il/uwieder, 2002.

[24] Zheng Zhang, Shu-Ming Shi, and Jing Zhu, "Somo: Self-organized metadata overlay for resource management in p2p dht," in *2nd International Workshop on Peer-to-Peer Systems (IPTPS'03)*, Berkeley, CA, 2003.

[25] Venkata N. Padmanabhan, Helen J. Wang, Philip A. Chou, and Kunwadee Sripanidkulchai, "Distributing streaming media content using cooperative networking," in *Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*. 2002, pp. 177–186, ACM Press.

[26] Miguel Castro, Peter Druschel, Anne-Marie Kermarrec, Animesh Nandi, Antony Rowstron, and Atul Singh, "Splitstream: high-bandwidth multicast in cooperative environments," in *Proceedings of the nineteenth ACM Symposium on Operating Systems Principles*. 2003, pp. 298–313, ACM Press.