A Robust Aggregation Tree on Distributed Hash Tables

Ji Li JLI@CSAIL.MIT.EDU
Dah-Yoh Lim DYLIM@CSAIL.MIT.EDU
MIT Computer Science and Artificial Intelligence Laboratory, 32 Vassar Street, Cambridge MA, 02139 USA

1. Introduction

A Distributed Hash Table (DHT) is a distributed resolution mechanism that manages the distribution of data among a changing set of nodes by mapping keys to nodes. A large number of DHTs have been proposed, and are expected to eventually become the fundamental components of many large scale distributed applications in the near future.

As P2P networks grow in popularity, there is an emerging need to collect aggregate information from such systems on the parts of both administrators and application users, such as available network storage, computation capabilities, and network size estimation for randomized DHTs. 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.*

We propose a bottom-up approach to build a robust tree in a P2P environment to implement aggregation. Our approach uses a parent function to help a node determine its parent dynamically, and maintain it using soft-state. Compared with previous systems on constructing trees in P2P networks (El-Ansary et al., 2003; Ratnasamy et al., 2001b; Rowstron et al., 2001) which are tightly coupled with their underlying P2P networks, our approach is independent of the underlying P2P topology and thus more flexible in constructing trees based on different requirements.

2. Parent Functions

Parent function plays a central role in our scheme. In this section, we define parent function and present an example of parent function.

2.1 Distributed Hash Table Overview

After the success of Napster at file-sharing, many other peer-to-peer systems have been proposed, especially those based on DHTs, such as CAN (Ratnasamy et al., 2001a), Chord (Stoica et al., 2001), Tapestry (Zhao et al., 2001), and Pastry (Rowstron & Druschel, 2001). DHTs share several common properties. First, many DHTs use circular and continuous namespaces. Second, they all provide efficient lookups. Most DHTs can resolve a lookup in $O(\log n)$ or fewer steps. Third, they are relatively resilient to node failures in that they automatically repair the network when nodes leave or fail.

With the three features, continuous namespace, efficient lookup and robustness, our goal is to build an efficient and robust aggregation tree in a dynamic environment. In this section, we define a parent function and propose a sample which is used in our work.

2.2 Parent Function

A key part in our work is a many-to-one function, $P(x)$, to map each node to a parent node in the tree based on its $id\ x$. 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.

Generally, $P(x)$, which we call a *Parent Function*, is a function that satisfies the following conditions:

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

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

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

where α is an id owned by the root of the tree, x is a node id, and $Distance(x, \alpha)$ is the distance between x and the root α. If a function $P(x)$ satisfies the above conditions, there is a directed path from all nodes to the root.

Note that a DHT namespace is much larger than the normal network size. Therefore, a node in DHTs is usually responsible for an id range. Like in many DHTs, we assume a node is responsible for the ids 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 definition, the node represents the id $P(x)$.

2.3 A Sample Parent Function

Parent functions play a central role in determining tree properties. The following is an example adopted in this work.

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

where k is a parameter that determines the branching factor of a tree, and $m = 2^{\text{address\_space\_bits}}$.

As shown in Figure 1, a tree resulting from this function is rooted at a node owning the $id\ \alpha$. The expected height of a spanning tree constructed with this function is $O(\log_k n)$, where n is the network size. The expected branching factor is approximately k if the nodes are uniformly distributed in the namespace (except for the root).

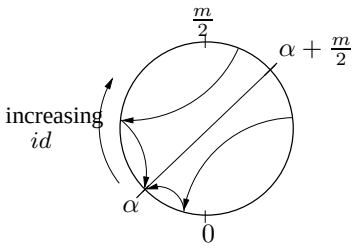

Figure 1. Aggregation pattern of the parent function. The circle represents the name space, and the arrows show the directed edges from a child to its parent.

2.4 Discussion

We believe that, due to the scale and dynamics of DHTs, a good parent function should also have the following features to be efficient and flexible.

First, it should guarantee that each parent node has approximately the same number of children, given nodes are uniformly distributed in the namespace. This helps to build a balanced tree. Our sample parent function satisfies this property: each non-leaf node has on average k children except the root.

Second, it should guarantee that node joining and leaving will not greatly affect the structure of an established tree, especially tree height. We identify that two factors make it hard to stabilize a tree. First, $P(x)$ usually maps to the node that follows $P(x)$. Another factor is that parent changes due to nodes joining or leaving may change the tree height. Our sample parent function satisfies this property because neighboring nodes are at the same level or an 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.

Third, a parent function should support branch balancing. Although ideally a good parent function can balance the number of children each node has, some nodes may be assigned too many children to handle due to the dynamics of peer-to-peer networks and variance in node distribution. We propose two branch-balancing schemes: admission control and dynamic adaptation. The problem is how the refused or abandoned children find their new parents. In the sample parent function, 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. This method 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.

3. A Bottom-up Tree Building Algorithm

In this section, we describe a bottom-up tree construction algorithm, which consists of two parts: tree construction and tree maintenance.

3.1 Tree Construction Protocol

The tree construction protocol describes how a node joins an existing tree as follows. Figure 2 shows an example.

1. When a new node x joins the network, as most peer-to-peer networks assume, it should know some node already in the network from which node x can set up its state. It is also from the introducing node that node x learns about the parent function, $P(x)$, and the root $id\ \alpha$.

2. Node x determines 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.

3. 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. After finding node y, node x sends a message containing $P^i(x)$ to y to register itself as y's child.

4. After receiving x's register message, node y will add node x to its list of child nodes together with the re-

ceived $P^i(x)$. If node y already has too many children to handle, it will use some admission control to redirect node x to other nodes.

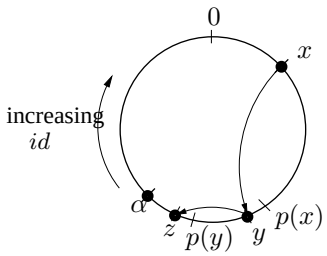

Figure 2. The node joining procedure. Node y is node x's parent in terms of $P(x)$. Node z is y's parent in terms of $P(y)$.

3.2 Tree Maintenance Protocol

After a tree is constructed, it needs to be maintained carefully in the peer-to-peer environment. The tree maintenance protocol is as follows.

1. After a node x joins the tree, it is henceforth x's responsibility to contact its parent node y periodically to refresh its status reliably as a child node. 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 fails, 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.

3. 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 observe it and 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. Figure 3 shows an example. 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 parent, if it knows that z' is between $P^i(x)$ and z. This recursive procedure continues until a proper parent is found.

4. Node x may notice that it should change the parent. This happens when x's current parent is found in terms of $P^i(x)(i > 1)$, 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 at its former parent. Figure 4 shows this case.

5. 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.

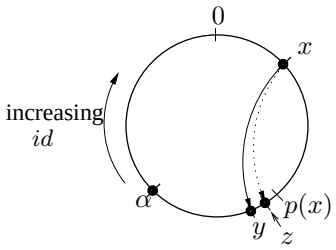

Figure 3. The first case of parent change due to node joining. Node y is node x's current parent in terms of $P(x)$. Node z is a new node which joins and covers $P(x)$. Thus z should be x's new parent. y can easily discover this by observing that z becomes its neighbor.

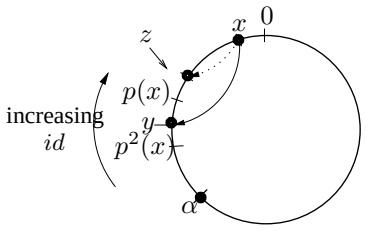

Figure 4. The second case of parent change due to node joining. x's current parent is y, in terms of $P^2(x)$, because there is no nodes between x and $P(x)$. Later, z joins and takes over $P(x)$ and thus should be x's new parent. x can easily discover this by observing that z becomes its neighbor.

3.3 Discussion

Our goal is to maintain it as a light-weight aggregation infrastructure, and as a base to construct other special-purpose and more complicated trees, such as media streaming. To keep it simple, we do not consider factors such as bandwidth or proximity. Since the refreshment message is a null message, we can use it to aggregate some general network information. For example, we can piggyback the size of the subtree in refresh messages, so that the root will get the size of the whole peer-to-peer network. Then the root and each parent node can piggyback the network size to the acknowledgments, so that in the end each node will learn the network size.

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 assigned 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 performed easily. As explained in Section 3.2, there are two cases when a node should change its parent. Both cases can be detected by a child or a parent via simply observing its neighbor change. Figure 3 and 4 show the two cases, respectively. Therefore, tree maintenance on parent changes is very cheap.

4. Performance Evaluation

We use Chord (Stoica et al., 2001) as the underlying DHT. Our experiments are divided into discrete time periods. In each period, a number of nodes join the network according to a Poisson distribution. At the point when a node joins, its departure time is set according to an Exponential distribution, and nodes are removed from the network when their lifespans expire.

As an example of usage, we estimate the available network storage in a P2P network. The parent function is the sample one in Section 2.3 in which α is set to $\frac{m}{2}$, and k is 4. The storage on each node keeps changing according to an approximately Gaussian distribution with a mean of 50MB and a standard deviation of 20MB.

Figure 5 shows the evolution of the network storage and the network size, and their estimates aggregated at the tree root with a node failure rate of 10%. The spikes in the figure are caused by the failure of intermediate nodes high up in the tree. The results demonstrate however that our algorithm recovers rapidly from failures, and the average estimation is very close to the true value.

5. Conclusion and Future Work

The aggregation problem is complicated by the scale and dynamics of P2P networks. We propose to build a robust tree over DHTs as a general light-weighted utility and as a base to construct trees for special usage. The reason for separating solution to tree construction from the underlying P2P topologies is to concentrate on a general purpose capability independent of but needed in many DHTs. The major advantage of our bottom-up algorithm is its relatively low overhead, resilience to node failures, and flexibility. We

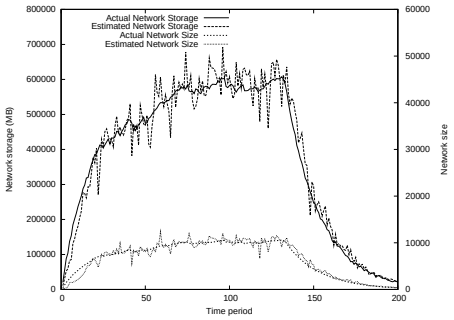

Figure 5. Network storage and size estimation.

are currently improving our approach by designing parent functions with different properties, and methods to overcome single points of failure in a tree. We are also working on comparison between our performance and other aggregation methods and tree-based broadcast schemes in peer-to-peer networks.

References

El-Ansary, S., Alima, L. O., Brand, P., & Haridi, S. (2003). Efficient broadcast in structured P2P networks. *IPTPS'03* (pp. 304–314). Berkeley, CA.

Ratnasamy, S., Francis, P., Handley, M., Karp, R., & Shenker, S. (2001a). A scalable content addressable network. *Proceedings of ACM SIGCOMM 2001*.

Ratnasamy, S., Handley, M., Karp, R., & Shenker, S. (2001b). Application-level multicast using content-addressable networks. *NGC 2001* (pp. 14–29).

Rowstron, A., & Druschel, P. (2001). Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)* (pp. 329–350). Heidelberg, Germany.

Rowstron, A. I. T., Kermarrec, A.-M., Castro, M., & Druschel, P. (2001). SCRIBE: The design of a large-scale event notification infrastructure. *NGC 2001* (pp. 30–43).

Stoica, I., Morris, R., Karger, D., Kaashoek, F., & Balakrishnan, H. (2001). Chord: A scalable Peer-To-Peer lookup service for internet applications. *Proceedings of the 2001 ACM SIGCOMM Conference* (pp. 149–160).

Zhao, B. Y., Kubiatowicz, J. D., & Joseph, A. D. (2001). *Tapestry: An infrastructure for fault-tolerant wide-area location and routing* (Technical Report UCB/CSD-01-1141). UC Berkeley.