Reducing Traffic and Delays in P2P Systems with Replicated Mutable Files

Andrew Rosen Brendan Benshoof Matt Erwin Anu Bourgeois
Department of Computer Science, Georgia State University
34 Peachtree St NW
Atlanta, Georgia 30303
rosen@cs.gsu.edu

Abstract—**Peer-to-peer networks generate an increasing amount of traffic each year as they become more widely used and as larger files become more common to share. This has led research into how peer-to-peer networks can be used for tasks such as keeping mutable files across a network up-to-date and using other points in the network to store temporary copies of the file to ease the load on the server by diverting traffic. The integrated file replication and consistency maintenance algorithm (IRM) combined the problems into one and examined the effects on a network using Chord. This paper proposes reducing the strain on the file owners by reducing polling and diverting polling traffic to the replica nodes. These changes can be accomplished without decreasing the hit rate at the replica nodes, while also decreasing the overall traffic in the network and average latency.**

I. Introduction

A. This doesn't belong here

Chord was designed as file sharing system as

Chord was modified into CFS for a distrubuted file storage.

II. Background

Not all peer-to-peer networks are equal; there are variety of protocols and methodologies that a networks could implement and this affects what kind of solutions are available to reduce the traffic on the network [?] [?].

The most basic type of network is a structured, centralized network. Peers in this network communicate with a central server to provide their files and to locate other peers that have the files they are searching for. This is structured in the sense that the layout of the overlay network is tightly controlled, in this case by the server(s). While this avoids the problems of routing, it has the same issues of scalability as a client-server layout and is not much of an improvement. An example of this network is the long defunct Napster [?] [?].

On the other side of the spectrum, there are unstructured, decentralized networks. These networks create overlay links between nodes in a random manner. This leads to a very unstructured overlay, but it is one that is very easily constructed. No single node is responsible for the whole of the network; files are located by sending out requests to neighboring peers, which in turn request from their neighbors and so on. Should a file become suddenly popular, this flood of this requests can easily bring some peers to their knees, unable to deal with the high level of traffic [?]. This makes these types of networks also a poor choice for implementation [?].

Modern P2P implementations are hard to classify easily, due to the variety of methods used to create a working network that avoids the weaknesses of the two above network types. Many networks today use a decentralized structured approach to distribute files, where the topology of the overlay is constructed and controlled by the protocol and the information about the network is distributed among the peers. This distribution is typically accomplished by a distributed hash table (DHT). Networks that use a DHT choose specific peers in which to place information about how to find particular files or data. These peers are chosen so that the peer's ID in the network corresponds to the file or data's ID, typically by hashing both ID's and comparing them [?].

In addition, the network topology is distributed among various peers. Each peer has a table consisting of other peers in the network and the means of communicating with them. The contents of this table are also controlled by the protocol. The table handles the routing of requests from one node to another; when a peer receives a request it cannot fulfill, such as information about where to find a particular file, it directs the the request to the node that is "closest" to the destination of the request. How this works is determined by each protocol [?]. It should be noted that closeness is relative to the algorithm; depending on what identifiers are assigned, a node in New York City might be "close" to a node in Russia, but "far" from a node physically a few miles away. Examples of protocols that use these techniques are trackerless BitTorrent [?], Chord [?], and Kademlia [?]. As our work is implemented using Chord, we provide more detail in the following section.

III. CHORD

The Chord protocol [?] takes in some key and returns the identity (ID) of the node responsible for that key. These keys are generated by hashing a value of the node, such as the IP address, or by hashing the filename of a file. The hashing process creates a m-bit hash identifier.

The nodes are then arranged in a ring from the lowest hash-value to highest. Chord then takes the hashed files and places each in the node that has the same hashed identifier as it. If no such node exists, the node with the first identifier that follows this value. This node responsible for the key κ is called the $successor$ of κ, or $successor(\kappa)$. Since we are dealing with a circle, this assignment is done in module 2^m space. For example, if there were some portion of the network with nodes 20, 25, and 27, node 25 could be responsible for the files with the keys (21,22,23,24,25). If node 25 were to decide to leave the network, it would inform node 27, who would then be responsible for all the keys node 25 was covering. An example Chord network is drawn in in Figure **??**.

With this scheme, we can reliably find the node responsible for some key by asking the next node in the circle for the information, who would then pass the request through the circle until the successor was found. We can then proceed to directly connect with the successor to retrieve the file. This naive approach is largely inefficient, and is a simplification of the lookup process, but it is the basis of how Chord theoretically works.

To speed up the lookup time, each node builds and maintains a *finger table*. The *finger table* contains the locations of up to m other nodes in the ring. The ith entry of node n's *finger table* corresponds to the node that is the $successor(n + 2^{i-1})$ $mod\ 2^m$ [1].

A. Stabilize

When a node n is told to find some key, n looks to see if the key is between n and $successor(n)$ and return $successor(n)$'s information to the requester. If not, it looks for the entry in the finger table for the closest preceding node n' it knows and asks n' to find the successor. This allows each step in the to skip up to half the nodes in the network, giving a $\log_2(n)$ lookup time. Because nodes can constantly join and leave the network, each entry in the table is periodically checked and updated.

To join the network, node n first asks n' to find $successor(n)$ for it. ,but those nodes won't acknowledge n's presence in the ring yet[2] . Node n relies on the stabilization routine to fully integrate into the network.

B. CFS

IV. MapReduce

At its core, MapReduce [**?**] is a system for division of labor. A task gets sent to a master node, who then divides that task among slave nodes, which may further divide the task. This task has two distinct parts: Map and Reduce. The map portion

A. Summary

Summary of MapReduce [**?**]

1) Differences with Hadoop: Hadoop is this an open source thingy for MapReduce [**?**]

[1]Because hash values won't be perfectly distributed, it is perfectly acceptable to have duplicate entries in the *finger table*.

[2]This is fine. You don't have to be *in* the ring to interact with it. You just have to be able to talk to a node in the ring.

B. Map

C. Reduce

V. CHRONUS

A. Implementation

Rather than the traditional tree structure, Chord takes advantage of the inherent ring structure.

VI. Experiments

A set of experiments were run on large groups of Amazon EC2 Micro Instances[**?**].

VII. Results

VIII. Conclusion