6.824 Final Project

Alap Sahoo (asahoo)
Edward Fan (edwardf)

May 11, 2018

1 Introduction

Online forums like Reddit are some of the most heavily used sites on the Internet today, but worries about administrative or governmental overreach are pervasive due to their centralized nature; a small number of entities can edit, delete, or frandulently create posts. A decentralized system can provide stronger authenticity and verifiability guarantees.

We implement a distributed forum system that uses a torrent-like approach to publicizing and distributing posts. When a post is created, a *seed* is propagated to every node in the network; any node can use the seed to download the post from some peer in the network. We explored, implemented, and tested two approaches to distribution: one similar in nature to breadth-first search, and one that coupled a distributed hash table with a gossip protocol.

2 Design

2.1 Requirements and Alternatives

In our original idea, we proposed immutability and verifiability guarantees: any node could confirm that a particular user made a particular post, and that posts could not be deleted or modified by anyone once posted. We initially explored using a blockchain to accomplish these tasks, but moved away from the idea once we realized there were fundamental scalability issues: every node would have to store at least a hash for every post, and bringing new nodes online would take prohibitively long.

We then considered a torrent-like approach. Fundamentally, there are two parts: seed distribution (a "front page" like structure to notify users of new posts) and post retrieval (to actually read content). There are many advantages to using such a framework; nodes can use the forum without having to store any posts, no global order of posts or comments is required, and special

nodes (like "archival nodes" that store all posts) are easy to implement. After implementing a small broadcast protocol to verify that such an approach worked, we implemented two separate approaches: one involving BFS, and one involving a DHT.

We were able to get immutability and verifiability very simply; every post includes a hash of the post and the poster's public key, along with a signature (and corresponding public key). The hash and signature can be easily verified. Although we no longer have as strong of a guarantee against deletion, both of the protocols described below do make it difficult for a post to be completely deleted from the network.

2.2 Breadth-First Search

Our initial implementation simply connected all nodes and had them broadcast all updates to each other. A logical extension, then, would to be to implement the same idea, but use non-fully connected nodes and distribute requests with a different scheme.

In this system, each node is connected to approximately $\log n$ peers, where n is the total number of nodes. Although we did not formally analyze the probability, it makes sense that all nodes should be in the same connected component; the probability of multiple components is vanishingly small since peers are chosen at random (through a sufficiently long random walk). When a node sends a post, it is passed to all of its peers, which in turn recursively forward it to its peers (hence the similarity to BFS). This leads to approximately $n \log n$ messages being sent for each post; although this high bandwidth usage is the primary downside of this scheme, it does have the advantage of not having hotspots, as every node will send and receive about $logn$ of those messages.

Whenever a node receives a post, it can choose to store the post or just the seed. This allows for significant configurability - a node that is low on storage can choose to not store any posts, getting everything over the network; alternatively, a node that wants to archive (and distribute) all messages can choose to do so. Any rule can be used; a node might choose to save posts from a certain user, that contain certain keywords, or any other scheme devisable from a post's content. In practice, storing a small constant fraction of posts is a reasonable default for most nodes.

Since posts can be stored anywhere, some assistance is required to efficiently retrieve posts. To aid in retrieval, as a post is propagated, any node that stores the post piggybacks the fact that it has stored it on the post. For each seed, each node keeps track of a node that reported storing the linked post. During retrieval, the cached node is contacted first; if it no longer has the post (for example, because it deleted some of its old stored posts), it can still provide forwarding information to another node that reported storing the post. As a

fallback, a node contacts its peers to look for routing information. Although this scheme does not provide the strongest deletion guarantees (it is possible that a node can fail to find a post despite it still existing on some other node), in the vast majority of practical situations, routing succeeds immediately from the cache. Additionally, one advantage of this scheme is that once a node retrieves a post, it typically will also have the post stored, allowing it to respond to peers or other nodes that query it for said post. This means that popular posts become easier to retrieve, an excellent quality.

The BFS implementation currently bootstraps new nodes by simply connecting them to a set of random peers and copying over a table of seeds. In practice, seed transfer should be done through a separate system, like the gossip protocol described in the next section.

Aside from high network bandwidth, the biggest flaw of the BFS protocol is that it requires at least a moderately high percentage of nodes to be online in order for routing and propagation to succeed. Although perhaps not entirely impractical for some sets of users, in a more general Reddit-like userbase, this requirement would require a level of abstraction that maps users to almost-always-connected peers (on some server) to allow for good mobile connectivity. The DHT implementation, described in the upcoming section, effectively solves this problem.

2.3 Distributed Hash Table with Gossip Protocol

The BitTorrent model that we draw inspiration from is not fully decentralized - trackers are traditionally a file stored locally at one source, and websites like PirateBay are generally necessary to learn the existence of torrents. However, BitTorrent clients have over the years addressed both of these issues. "Trackerless" torrents that utilize Distributed Hash Table (DHT) implementations allow for distributed trackers; the Mainline DHT employed by BitTorrent is the largest DHT in the world. The problem of decentralized torrent search has been addressed by Tribler and other clients, which utilize a gossip protocol (GP) to propagate knowledge of different torrents through a network. Both of these solutions map nicely to one of the key problems in implementing a decentralized forum - storing/retrieving posts, and maintaining a "front page" of seeds to learn about them. Here, we discuss our design and test implementation of a DHT-GP system for propagating and storing a decentralized Reddit.

Seeds are propagated by all nodes in a system to their peers using a gossip protocol - nodes will randomly pair off with other nodes, which exchange information about the last S most recently received seeds and store new seeds that they receive from their partner. Seeds are lightweight - less than 100 bytes in size - so many seeds can be conveyed per gossip session and be stored locally. Furthermore, the gossip protocol does an excellent job of conveying information across a large network. A seed on one node will be propagated to all nodes in

a network of size n in $O(\log n)$ time; assuming that each node gossips once per second with random peers, it takes less than 20 seconds to complete propagation. While the ideal model of completely random peers and constant gossiping cannot exist in practice, even an approximation is enough; the Bitcoin network uses a similar approach with eight known peers per node and converges quickly. (We referenced a survey paper by Alberto Montresor - http://disi.unitn.it/ montreso/ds/papers/montresor17.pdf - in examining gossip protocols.)

While large number of seeds can be stored by everyone and therefore passed along by anyone, posts cannot be due to their size. Therefore, certain nodes can choose to identify as "storage nodes" and claim responsibility for storing and answering queries about posts. Storage nodes are organized using a system similar to a DHT. There are M hash slots in this DHT, and each storage node chooses some value m. Posts are stored in the slot given by the last $\log(M)$ bits of their seed hash; a storage node with value m claims responsibility for as many of the posts mapped to m as it can. Storage nodes in each level form a network to facilitate the communication of posts, which are also exchanged in a gossip protocol, trading the last P posts at random. While posts take more time to transmit than seeds, there are far fewer nodes in each m-level network, so convergence across the network is rapid. Finally, storage nodes can communicate with other storage nodes outside their layer by maintaining a set of peers in the layer above its own. This forms a ring-like structure in the DHT's layers; a more sophisticated approach might use more skip pointers (like Chord), but we don't believe that's necessary for our network.

To retrieve posts, each node maintains a table of up to M different storage nodes (one per layer). To post or access a post associated with layer m, a node finds the cached storage node in that layer. If it lacks such a storage node or if the relevant storage node is unresponsive, it can simply query the storage node at layer $m-1$ and ask that storage node to check its "above" neighbors. If the node at layer $m-1$ also does not respond, the node can simply recurse even further down - as long as there's one valid connection, it is able to access any point in the hash table. While in the worst-case $O(M)$ time is required to find a node on the correct layer, in the average case only a constant number of lookups is necessary, as each node finds and retains a reliable node per layer. A node can post by sending its message to the relevant storage node, waiting some amount of time for the message to be propagated throughout the m-level network, and then adding the message's seed to its list of seeds and allowing the GP to perpetuate the information. Retrieval simply involves asking the relevant storage node about a given seed and verifying the result.

Certain heuristics and properties help the system deal with node failures. For example, by removing storage nodes from cache that fail to provide a post, nodes will eventually converge on "reliable" nodes for each layer. A storage node in layer m will better maintain "reliable" connections to storage nodes in layer $m+1$ by asking any of the nodes querying it for the node stored at $m+1$ in the

querying node's hash table, and incorporating that information - the querying nodes will have a good idea as to which storage nodes are useful. Finally, nodes exchanging seeds can simply prune peers that they never get useful seeds from.

The DHT-GP system should scale fairly well; for example, assuming both the seed and post sharing gossip protocols can occur in under 2 seconds, a network of 1 million nodes with 250,000 storage nodes scattered across 250 hash slots should be able to store a message across the 1000 relevant storage nodes and perpetuate the message's seed across the full network in under a minute each. (Simulations show that a GP can push an item across a 10^6 size network in 15-17 gossip iterations, and across a a 10^3 in 6-7.) Retrieval should be equally fast - while worst-case requires marching down the full M length hash table and then checking with several storage nodes, the average-case scenario involves pinging a single or a few reliable storage nodes. The design suits the decentralized environment well, since both storage nodes and regular nodes are not required to hold onto any particular posts or seeds for the system as a whole to function. Finally, network traffic is minimal for new posts, since the full data of the post is only transmitted within a small subset of nodes, not across the whole network.

Beyond being relatively complex, the main issue with this system is that the main parameters - M, P, and S - have to be defined ahead of time, and can cause problems if not chosen correctly. Ideally M should scale with the number of users in the network If it's too big, i.e. $M = 1000$ for a network of only 100 users willing to store, the users acting as storage will have to run multiple storage nodes on a single instance across a range of m values. If it's too small, i.e. $M = 100$ for a network of a 10^6 users willing to store, each hash slot will be overloaded with nodes and slow down the GP. P and S - the parameters that define how recent a post or seed can be and still be shared by the GP - need to be large enough such that no information stops being gossiped about until it's spread to most of the network. These, again, likely need to scale with the network. The other key issue is availability - the storage nodes might be unevenly distributed across the layers, or a single layer might get an overabundance of the requests, if, say, a popular post is stored there. The latter is partially mitigated by how we hash comment chains - a comment is stored in no relation to its parent, so popular pages should be spread across the DHT and assembled by the client. And both can be solved in large part by simply allowing storage nodes to pay for retrieval and storage. In the long run, torrents demonstrate that the cost will tend towards 0, since the only thing necessary to undercut another storage node is a copy of its data. And an economic incentive will create a reason for storage nodes to "behave", by properly maintaining posts and spreading themselves across storage nodes appropriately, while preventing meaningless spam from nodes querying the DHT.

3 Implementation

3.1 Code Structure and Tests

The code is available at `https://github.com/edfan/dreddit`.

`src/dreddit` contains the primary implementation. The shared post and verification infrastructure is located in `reddit.go`, while the network backends exist in `broadcast.go`, `bfs.go`, and `dht.go`. A simple script to show how fast a GP can push information through a network can be found at `basic`$_{g}p_{s}imulation.py$.

Tests for the BFS and DHT implementations are located in bfs_test.go and dht_test.go respectively. Running go test (perhaps after go get) should successfully run all tests for both backends, as well as a simple verification test. The tests include massive concurrent posting, recovering from disconnected peers, and deletion of posts from some nodes (to simulate nodes emptying their caches).

labrpc is used as the underlying network for testing.

3.2 Live Demo

(This is a work in progress; will be online by the presentation.)

Using WebSockets, we're implementing a simple browser-based demo where users can control nodes in a pre-built network. Some preliminary proof-of-concept code is in src/websockets, although no real UI exists (and there are plenty of bugs).