TinyTorrent: Implementing a Kademlia Based DHT for File Sharing

A CS244B Project Report By

Sierra Kaplan-Nelson, Jestin Ma, Jake Rachleff
{sierrakn, jestinm, jakerach}@cs.stanford.edu

***Abstract*—We implemented a distributed peer-to-peer file sharing service called TinyTorrent. Users discover other users who store files on their system through Kademlia, a well known peer-to-peer distributed hash table. In this paper, we describe the architecture and implementation of TinyTorrent and Kademlia. Our approach and implementation are motivated by providing a readable and fast distributed system.**

I. Introduction

Some recent papers in computer science have focused on readability and understandability [1]. This benefits students and novice programmers who aim to implement large, complex distributed systems. However, when researching such systems online, existing implementations are bloated with unnecessary complexity and length. When readable implementations do exist, they often are in Python, JavaScript, or other inherently slow, non-systems languages. We believe that it is in the best interest of all programmers building distributed systems to have simple, lightweight, yet performant implementations of important algorithms in fast, systems-focused languages.

One concept in vogue the start of the 21st century was the distributed hash table [2]. At its simplest, this data structure supports two simple operations for a client: put, and get. Put allows the client to insert a key and value, and get allows the client to get the value for a given key. Externally, it appears like a regular hash table, but in reality, keys are distributed across multiple machines. Through distribution, DHTs prevent single node overload, and have strong applications in load balancing and content distribution [3]. During the rise of peer-to-peer file sharing, one distributed hash table in particular, Kademlia, became popular due to its guarantees of performance, consistency, message efficiency, and simple algorithm [4]. It serves as the underlying protocol and network for BitTorrent, the popular file sharing service.

Cursory searches for easy-to-understand and well-architected Kademlia implementations on Github and other repository-hosting services appear unfruitful. Though the algorithm described in the original paper is relatively straightforward, many implementations online are either dense and abstruse or written in languages trading ease of coding for a loss in performance. In this paper, we describe our implementation of Simple Kademlia, a fast, bare bones C++ library implemented with Boost Asio and Cereal. This serves as a simple, readable example of an integral distributed system in C++. Further, we implement a file sharing application, TinyTorrent, on top of SimpleKademlia. Through TinyTorrent, we both aim to both show the power and usefulness of SimpleKademlia, and provide a readable yet performant Kademlia library for users and learners to understand the complex nature of asynchronous, threaded, network programming in C++.

II. System Overview

A. System Architecture

Our system consists of two parts - TinyTorrent and SimpleKademlia. The interaction of each part can be visualized in Figure 1.

Each node that joins the file sharing network boots up TinyTorrent, launching both a client and server. The user requests or publishes a file through the client. To publish a file, the client can use the Kademlia API to publish the file to the network. Post-publishing, nodes connected to the Kademlia

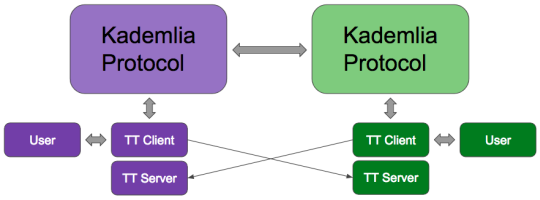

Fig. 1. High level architecture of TinyTorrent service. Users interact with a client interface (command line). The client interacts with the Kademlia protocol using a simple API. Kademlia nodes interact with one another through RPC messages. Clients communicates with remote servers.

network are able to request from Kademlia the node(s) which are hosting the desired file. To request the location of a desired file, the client uses the Kademlia API to get the file by specifying its name (ideally, by specifying some better identifier of the actual data like a checksum or hash). The client will be returned the IP address and port of another server in the network that has that value. The client is the only interface to Kademlia from TinyTorrent. Once the client receives the proper IP address and port pair from which to download the desired file, it contacts the server to which that pair belongs. In the case that multiple nodes are hosting the same file, the client will possibly be returned a list of peers to contact in case one fails.

Kademlia is structured to have its own protocol consisting of a set of RPCs and core algorithms - all Kademlia interactions are internal to Kademlia, and do not respond to the client unless an API call completes. Currently, SimpleKademlia is a library that could run separately from TinyTorrent. This decision was made to allow other application creators to fork our SimpleKademlia code separately from TinyTorrent and design a different application taking advantage of a Kademlia DHT.

III. Kademlia DHT

The Kademlia DHT we built from scratch in C++ maps all node IDs and keys to a 256-bit ID space. Node IDs are computed by taking the SHA-256 hash of a peer's network information. Each node in the DHT consists of a Protocol object and a Network object (see figure 2). The protocol is responsible for housing a portion of the DHT's key-value pairs and delegates I/O operations for reading/writing to a separate thread. The network engine is responsible for external communication

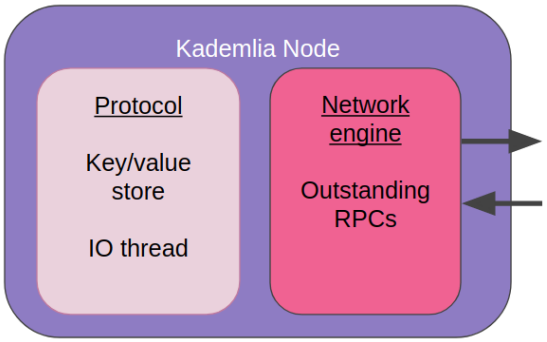

Fig. 2. Internal architecture of a Kademlia node. A node consists of the protocol managing data and I/O operations, and the network engine responsible for sending and receiving RPCs.

with other node's network engines through RPC messages. The network engine maintains at all times a collection of outstanding sent RPCs uniquely identified with a request ID.

A. Kademlia API

The Kademlia DHT offers applications a simple API:

- `node(ip, port)` - Instantiate a local Kademlia node on the given ip:port
- `bootstrap(peer)` - Connect to the network through a pre-known peer. This peer should be known in advance as a bootstrap node for any node to join the Kademlia network. (Explained in III-E)
- `get(key, callback)` - Retrieves from the network a list of peers that contain the value associated with the key. A client can then connect to these peers to download the value. How peers exchange values is up to the application.
- `put(key)` - Puts the key in the DHT, meaning the application notifies the network of nodes it contains the value for the key. Multiple peers can put the same key (assuming the same value). Subsequent calls to `get(key, ...)` potentially retrieve a list of peers storing the value.
- `join()` - Disconnects the Kademlia node from the network and stops communication with remote peers.

B. Routing Table

Each Kademlia node keeps track of its known contacts in a routing table, which is a tree where each leaf node contains a k-bucket. A k-bucket is a list of k contacts sorted by most recently inserted. A routing table can contain up to 256 k-buckets, which is the number of bits in our ID's. The procedure of inserting into our routing table follows the same algorithm detailed in the Kademlia paper [4]. When inserting a new contact into a full bucket, if that bucket contains the current node's own id it is split and its contents are divided amongst the two new buckets and insertion is attempted again. Otherwise the contact is dropped because it is not sufficiently close to the current node. Unfortunately we did not have time to finish the refresh buckets procedure. The routing table can return the k or α closest buckets to a given key.

C. Messaging Protocol

Kademlia nodes communicate by exchanging 4 types of RPCs:

- `PING` - verify that a node is alive
- `STORE` - store a key-value pair at the destination node. The recipient will record that the sending peer contains the value for a given key.
- `FIND_NODE` - requests at most k nodes from the recipient's routing table which are closest to the given key. This primitive RPC is used in the general lookup algorithm.
- `FIND_VALUE` - returns the corresponding value for a given key if the recipient has stored the requested key. Otherwise, functions as a `FIND_NODE` RPC.

A node is represented as `NodeInfo` object which, at the very least, contains the fields `[IP address, UDP port, nodeID]`. The 256-bit node ID is the SHA-256 hash of the IP address and port, so node IDs have a very low probability of colliding. If duplicate node IDs is ever an issue, nodes could also verify each other's identity by checking the packet's IP header. `STORE` messages store `NodeInfo` objects as values, and both `FIND_NODE` and `FIND_VALUE` return a vector of `NodeInfo` objects.

Senders keep track of all outstanding RPCs, each identified by a unique TID. Requests are resolved upon receiving response messages with corresponding TIDs (responses with unfamiliar TIDs are discarded). The message type hierarchy is represented polymorphically, meaning one can easily add a custom RPC by introducing a new message class and specifying how to process requests and responses.

D. Lookup Algorithm

In order to provide `get` and `put` functionality, Kademlia uses a lookup algorithm to locate the K closest nodes to a key. The put function stores the key, value pair on each of the K closest nodes. The get function either terminates when it finds the value or finds the K closest nodes to a key and none of them contain the value. This lookup algorithm leverages the `FIND_NODE` or `FIND_VALUE` RPCs and sends out RPCs in waves, in parallel and asynchronously [4]. The lookup procedure, originally described recursively, is described iteratively as follows:

1) Select α contacts from the closest non-empty k-bucket to the key being searched for. If there are fewer than α contacts in that bucket, Kademlia will select from buckets increasingly farther away from the key. α represents the number of parallel requests.
2) The initiator node sends asynchronous `FIND_NODE` (or `FIND_VALUE` RPCs in parallel to each of the α contacts.
3) Upon receiving each RPC response, the initiator node adds the received list of k-closest nodes to a sorted set, sorting the nodes using the XOR distance between a node's ID and the key.
4) The initiator repeatedly selects the first α contacts from the list that have not been contacted yet. `FIND_NODE` RPCs are sent to these contacts. At most α requests are in parallel at once, and the recursive procedure can begin before all requests from one round have returned. The α requests will always be to the closest nodes the initiator has heard of.
5) The search terminates when the k closest nodes to the key have been queried and responded, or when all the known nodes in the network have been queried. In the case of sending `FIND_VALUE` RPCs, the search terminates as soon as the initiator node receives a response that the value has been found, meaning the

RPC response payload contains a list of peers whom the initiator node can contact for the data. (We were unable to implement removing a node from consideration if it does not respond. This is in our future steps).

6) The list of K-closest nodes is passed to a callback; for `put`'s, Kademlia sends the nodes in the list `STORE` RPCs. For `get`'s, the list of nodes is passed to a callback which calls the user-specified callback with the list (if found) or an empty list if the value was not found.

E. Joining the DHT

A node that wants to join the network of existing nodes does so by connecting through a well-known peer during the `bootstrap(peer)` phase. After receiving a successful `PING` response from the bootstrap node, the joining node inserts the bootstrap node into its routing table (into k-bucket KB), and performs a lookup of its own ID. Because the only contact in its routing table is the bootstrap node, the joining node's k-buckets are subsequently populated with nodes between itself and the bootstrap node. After bootstrapping, a node's routing table should be sufficiently populated. The additional "refresh" routine was left out of the bootstrapping method due to the routing table and k-bucket's simplicity. The refresh routine allows the joining node to look up a random ID for each of the k-buckets farther away from the one housing the bootstrap node, thereby populating all k-buckets in its table.

IV. TinyTorrent Application Layer

A. Usage Overview

TinyTorrent, our client application, is built to handle many concurrent requests. To join our file sharing service, a user must run two processes: A TinyTorrent Client and TinyTorrent Server. For the sake of ease of use, the client and server are run and forked from one program. Clients provide an interface to the user to get and put data on the DHT (currently this interface is an interactive command line). The TinyTorrent client moreover initiates TCP connections to other TinyTorrent servers to request files.

The TinyTorrent server is simply a program that accepts and establishes incoming TCP connections to serve out files.

B. TinyTorrent Client

The TinyTorrent client serves as the main program between TinyTorrent and Kademlia, using the Kademlia API.

The TinyTorrent client's job is to allow a TinyTorrent user to publish and download files. To do so, the client reads commands from the user. When a user decides to publish a file, she enters the command: `put filename`. This simply calls the Kademlia node's put API, and returns to the user.

To download, a user enters the command: `get filename`. To remain non-blocking during heavy network IO, the client spins up another thread to handle all execution. First, the client must look up the correct list of nodes that contains the desired file in Kademlia. Once Kademlia returns the proper nodes, the client simply requests the file from the server, and subsequently downloads it over TCP. The client iteratively tries to connect to each of the remote nodes returned by Kademlia's get request until it can connect and download from one of them. If none are available, it returns without downloading.

C. TinyTorrent Server

The server is responsible for serving out files to incoming requests from remote clients. When a request comes in from a client, the server creates a new TCP connection to store information specific to the download from the client. The server must immediately spin off a thread to begin reading the desired file in from disk. The background thread reads in the file in small chunks that it queues up for TCP connection to process and asynchronously send to the client as chunks are read into memory. It is possible that a TinyTorrent user solely wants to issue `get` requests and not put any data on the network. The extra server process would not be required at all; this would count as a small extension of the TinyTorrent application if users could specify their status as a downloader, uploader, or both.

V. Software Implementation

The TinyTorrent client and server application and the underlying Kademlia DHT are implemented in C++ using Boost libraries for asynchronous I/O and network programming (Boost Asio). Messages are represented polymorphically for readability and extensibility. Message serialization

and exchanging is done over UDP using the Cereal serialization library. Apart from additional third-party libraries used for hashing, concurrency, and serialization, the underlying Kademlia DHT is 1500 lines of C++, and the client-server application is an additional 400 lines of C++. Currently, the client is configured to be able to submit concurrent requests for downloads equal to `std::thread::hardware_concurrency`, but could trivially be modified to allow a larger number of concurrent requests.

VI. Limitations and Future Work

In our implementation we focused on providing a simple, correct implementation of Kademlia in C++ with a client library demonstrating its capabilities. We had to make some assumptions to finish it within the class time period, making it less robust. Our major assumptions are that messages are sent successfully and a node never goes offline. In the future, we need to support RPC timeouts and message retransmission.

As mentioned in the lookup procedure section, we need to support removing nodes that do not respond quickly from the k closest nodes.

For bootstrapping, we need to add refreshing from bucket X to the farthest bucket.

The biggest thing we would like to add is timeouts on the key,value pairs. This would require a node looking through its storage and removing outdated keys on a daily basis. It would also require the client calling put on all its files at regular intervals to refresh the ttl of key,value pairs.

Finally, we would also like to remove contacts that have not responded to any requests for some amount of time.

References

[1] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proceedings of the 2014 USENIX Conference on USENIX Annual Technical Conference*, USENIX ATC'14, (Berkeley, CA, USA), pp. 305–320, USENIX Association, 2014.

[2] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *Proceedings of the 2001 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '01, (New York, NY, USA), pp. 149–160, ACM, 2001.

[3] E. Nygren, R. K. Sitaraman, and J. Sun, "The akamai network: A platform for high-performance internet applications," *SIGOPS Oper. Syst. Rev.*, vol. 44, pp. 2–19, Aug. 2010.

[4] P. Maymounkov and D. Mazières, "Kademlia: A peer-to-peer information system based on the xor metric," in *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, (London, UK, UK), pp. 53–65, Springer-Verlag, 2002.