Peer-to-Peer Wikipedia

Rain Gu and Nick Bradley

December 13, 2016

Abstract

We were motivated to create a peer-to-peer version of Wikipedia by our desire to remove proprietary interests in the online encyclopedia. Our prototype uses an enhanced version of Chord which includes replication and caching and the Treedoc CRDT to allow for disconnected editing of articles without any other form of concurrency control. Its design is scalable and robust to failures while maintaining article consistency and availability.

1 Introduction

Wikipedia is a free online collaborative encyclopedia funded and hosted by the non-profit Wikimedia foundation. In its current incarnation, Wikipedia is run in a typical client-server configuration giving Wikimedia full control of the content and infrastructure. To fund the administration costs, Wikimedia relies on public donations making it is quite possible that Wikipedia could become permanently unavailable. By moving to a peer-to-peer version, we remove proprietary interests in the system and eliminate costs associated with administrating infrastructure. Both approaches require direct support from users: in the case of Wikipedia it is a monetary donation; in our peer-to-peer system, it is storage and network resources. It is an interesting question whether users prefer the former or later but we do not attempt to answer it.

The primary goal of a wiki, which promotes cooperative editing of articles of text, is to maintain the consistency of the article while making it available for editing to as many users as possible. Once the number of users increases, copies of articles may need to be hosted on multiple servers to ensure availability. Here, we briefly compare the client-server approach of Wikipedia to the peer-to-peer approach of our system.

Support for collaborative editing of articles hosted by Wikipedia could be provided in several different ways. They could use a commutative replicated data type (CRDT) as in our prototype, or they could serialize operations through a master server using a consensus protocol like Paxos or Raft. Consensus protocols do not scale well, so they are most appropriate when used with a small number of servers.

To make its service more responsive, Wikipedia could either add more servers (scale out) or increase the resources available to its current servers (scale up). In our peer-to-peer version, we chose a protocol that is efficient in the number of messages it sends to ensure that the physical network can support many peers. However, we cannot directly increase the number of hosts (although peers must be willing to host articles if they want to view articles).

In this report, we detail our prototype peer-to-peer version of Wikipedia. It implements both an enhanced version of Chord [3] and the unoptimized version of Treedoc [1, 2]. Chord provides efficient article lookup by organizing the peers in the network. It handles peers joining the network as well as peer failure and churn. We base our articles on Treedoc which guarantees that different versions of an article converge without concurrency control. Together, these provide a scalable system which supports highly available, (eventually) consistent articles.

The remainder of this report proceeds as follows. Section 2 provides background on the Chord protocol and the Treedoc CRDT. Section 3 details the challenges of running a collaborative wiki on a peer-to-peer network and limits the scope of the problem. Section 4 discusses our prototype system. Section 5 evaluates the prototype. Section 6 concludes the report.

2 Background

In this section, we briefly review Chord [3] and Treedoc [1]. These are fundamental components of our prototype.

2.1 Chord

Chord refers to both a protocol and an algorithm for managing a distributed hash table (DHT). All entities

(peers and articles) are assigned an m-bit identifier[1], or *key*, which is obtained using a consistent hashing function such as SHA-1. The key represents the entity's position in the Chord ring. The key provides a mechanism for determining on which peer each article should be stored. Peers are identified by hashing their IP addresses while articles are identified by hashing their titles. Each peer in the system maintains the IP address of its *predecessor peer* (the peer immediately preceding it in the ring), its *successor peer* (the peer immediately following it in the ring) and a portion of the key-values in the DHT. The key-values that are stored on a specific peer are those whose keys fall between the peer's predecessor's key and its own key.

2.1.1 Lookups

When a peer wishes to retrieve a value from the DHT, it must do so by searching the network. Because of the Chord structure, lookups can be run in logarithmic time in the number of peers, N. When a peer initiates a lookup, it first checks to see if it or its successor is responsible for the lookup key. If not, it forwards the request to its successor which repeats the same process. This procedure repeats until the host peer is found, at which point its address is returned to the initiator. To reduce the search time, each node maintains a finger table consisting of m IP addresses which act as a routing cache. Specifically, the ith entry in the finger table on peer with ID n contains the ID of the first peer f that succeeds n by at least 2^{i-1}, which is the successor of $n + 2^{i-1}$. By maintaining the finger table, the lookup will "skip" to the closest peer in the table where it proceeds as before. This optimization reduces the total lookup time from $O(N)$ to $O(logN)$ with high probability.

2.1.2 Joins and Exits

A Peer, J, wishing to join the network needs to know the IP address of another peer already in the network. This peer determines J's position in the network and the IP address of its successor peer, S. Upon joining, J will notify S that it should be its new predecessor. Eventually, S's predecessor will stabilize[2]. Key-values on S that lie between J and its predecessor must be transferred to J. Once this completes, J is fully integrated into the network.

A peer can voluntarily leave the network by sending its key-values to its successor and then notifying its predecessor and successor of its departure. Upon receiving the notifications, the peers will initiate their stabilize procedures.

2.1.3 Failures

Peer failures are detected automatically during the stabilize procedure. When a peer's successor does not respond to stabilize requests, it is considered failed. To allow for recovery in this situation, each peer maintains not just one successor, but a list of r successors, where r is a measure of fault tolerance. Then, when a peer detects that a successor has failed, it will notify the next successor in the list, thus repairing the network structure. Unfortunately, the key-values stored on the failed node are lost. We address this issue next.

2.1.4 Replication

In the vanilla version of Chord, each key-value pair is stored only on a single peer. If that peer is unavailable, its key-value pairs are not available to any other peers and, in the case of failure, the key-value pairs are permanently lost. Replication of the key-value pairs is thus necessary to avoid this situation. However, it adds complexity to the design and implementation as we now need to consider the number, location, consistency, etc. of the replicas as well as the cost of maintaining the replicas.

There were several ways we could have chosen to implement replication. One way would be to replicate the key-value pairs to random peers in the system and, upon failure of the host peer, find the replica and put it on a new host peer (a peer in the successor list of the failed peer). This approach would require some mechanism for maintaining a sufficient number of replicas.

Instead, we decided to support replication by taking advantage of the peer's successor list: each successor stores replicas of all key-value pairs stored on the peer. This is a good method because it does not require maintaining any additional state about the system and it is controlled by the same parameter r as the successor list. This means that replicas are as fault-tolerate as the underlying Chord system.

Replication of a peer's key-value pairs occurs during maintenance of its successor list. The peer will push the key-value pairs to all of the successors on the list. In our prototype, we maintain consistency of the replicas using a CRDT which requires no consensus among nodes.

Now if a peer fails, its predecessor will notify the next available successor in its successor list as usual. This successor should have all of the key-value pairs that were originally owned by the failed peer, hence

[1]For m-bit identifiers, Chord supports $2^m - 1$ entities.

[2]Each peer will periodically run a stabilize procedure that will check if its successor's predecessor is still itself. If not, it will notify the new predecessor.

no pair is lost. If a peer joins the system, it will take over ownership of some replicated key-value pairs as explained in section 2.1.2. It is still possible that key-value pairs may be lost if multiple successors fail before the stabilize procedure runs (e.g. during a regional network partition) but this can be mitigated by selecting larger values of r.

2.2 Treedoc

Treedoc is a CRDT that was designed to support cooperative text editing. It allows users located at different sites to modify a shared article by operating on a local copy and then sending the changes to other sites to be replayed. Since Treedoc is designed such that concurrent inserts commute with one another, copies of the article converge automatically without the need for complex concurrency control. Treedoc is presented to the user as a linear sequence of paragraphs.

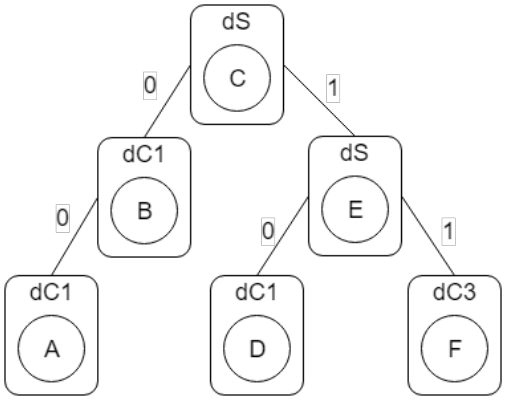

Figure 1: A basic Treedoc.

2.2.1 Identifiers

To provide commutative operations, Treedoc requires that every paragraph have an identifier and that the identifiers are unique, totally ordered and stable. Further, the identifiers must come from a dense identifier space such that it is always possible to generate a new identifier between any two existing identifiers.

Treedoc uses an extended binary tree to provide a dense identifier space: paragraphs are stored as nodes in the tree and the path to each paragraph acts as its identifier. In addition, nodes include the identity of the user who made the insert. Paths, represented as bit strings where 0 represents a left branch and 1 represents a right branch, are unique, stable and ordered by walking the tree in infix order. This is shown in figure 1: the circled letters represent the paragraphs and the letters prefixed by d represent the identity of the user who made the insert. For example, the leftmost node has text A, was inserted by dC1 and has id [0(0:dC1)]. It would be displayed as ABCDEF. Note that a more space-efficient implementation could be achieved by separating storage from identification (e.g. by storing the paragraphs as an array of (paragraph, path) pairs). The tree version would only need to exist to generate a new identifier for insertions.

A standard binary tree is not sufficient to represent concurrent inserts since different users may insert different paragraphs in the same position. To handle this case, the tree structure is extended by allowing each node to contain a list of "mini-nodes" (figure 2). In case of concurrent inserts along the same path, the paragraphs are inserted as a mini-nodes. However, this violates the total ordering of identifiers requirement since the position in the list may be determined by the order of execution. Therefore, we order the mini-nodes by the user's identity.

The algorithm for generating a new path for insertion is quite simple. Assume a new node is to be inserted between P and F, where P comes before F in infix order. If P is an ancestor of F then the new node should be inserted to the left of F. Otherwise, it should be inserted to the right of P.

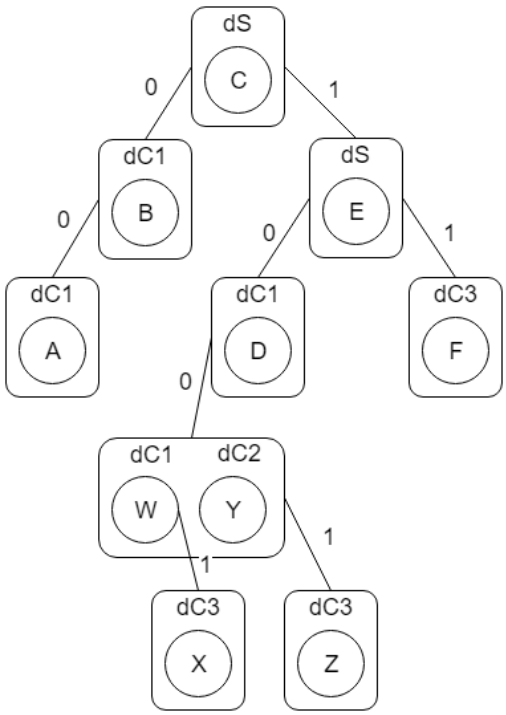

Figure 2: Treedoc with mini-nodes.

2.2.2 Operations

The Treedoc structure is modified by the two edit operations:

- `Insert(path, text)`, inserts the text into the tree at the specified path.
- `Delete(path)`, removes the paragraph at path. The paragraph is not actually removed from the tree as other sites may reference it in a concurrent replay. Instead, a tombstone flag, which controls visibility, is set.

Commutativity of the operations is guaranteed as follows. For any two operations that refer to independent identifiers, the operations commute since their effect on the tree is independent of execution order. By the identifier uniqueness requirement, new identifiers will be independent which implies two concurrent insert operations commute. In cases where concurrent operations act on the same identifier, we must have that an insert happens-before a delete so they can never be concurrent. Similarly, a delete can never precede an insert for the same identifier, so they can never be concurrent. Finally, the delete operation is idempotent so its effect will be the same regardless of the execution order. This shows that the operations are commutative. Thus, articles at remotes sites will converge if the operations are replayed in the same order as the modifying site.

3 Prototype Design

Deploying a service in a peer-to-peer environment presents many challenges because of the scale of the system and the typically unreliable nature of peers. For our prototype, we focus on availability and scalability and the consistency of articles.

In our system, availability means that once an article has been put on a host peer, any client peer can access it. This means that the article must be replicated on some other peers in the system so that the article will be accessible in the event the primary host peer is unavailable (i.e. disconnected or not running the Chord service). The version of Chord presented in [3] was designed to scale to the level needed to support internet applications.

We introduce a caching mechanism to improve the availability and scalability of the original Chord protocol. When a peer modifies an article, in addition to sending the changes to the host peer, the cache of every peer on the lookup path is also updated. When a peer wishes to retrieve an article, the lookup procedure will return a cached version unless the peer explicitly requests a non-cached version. Cached articles are deleted after a set time to ensure that they do not diverge too much. This mechanism increases availability since an article will be accessible on more peers. It adds scalability by reducing the number of lookup messages required to retrieve an article: the caching mechanism piggybacks on the existing lookup messages required to push an article.

Article consistency is achieved by using Treedoc CRDT which ensures that all versions of an article converge. It does not require concurrency control. This feature further increases both the availability and the scalability of the system since modifications do not need to be serialized which would add delay to the system. The automatic convergence of Treedoc makes our caching system trivial to implement. It should be noted that we only guarantee eventual consistency: once no more changes are made, all copies of the article will show the same content.

3.1 Design Considerations

We assume peers

- Are non-Byzantine since malicious peers could easily break the Chord protocol.
- Fail at a rate slow enough that the Chord structure can be repaired between failures.
- Have a unique address that is fixed once they join the network. Addresses are used as user identities when tracking changes to articles and therefore must not change.

Our prototype does not support deleting or remaining articles nor does it support substring- or content-based lookups. Deleting articles is a trivial problem since there is only one host peer: attempts to modify a deleted article would fail and cached versions would not be updated[3]. The cache expiration policy ensures that eventually all copies of the article are removed from the system. We simply ran out of time to implement this feature. As a workaround, users may simply delete all content within the article. Renaming articles is challenging since the article title is used as the lookup key. One potential solution would be to include an independent title field in the article structure. Unfortunately, this would hamper the simple elegance of Treedoc by requiring other forms of concurrency control. A workaround is to migrate content from an exiting article to a new article with the desired name. Finally, since we are using the title

[3] All replay logs would need to indicate whether they expect the article to exist so that deletions could be detected.

as the lookup key, it is not possible to directly support substring matches on Chord. A potential solution is to maintain article meta-data, for example its title and keywords, in the network. When performing a lookup, the meta-data would first be queried to obtain the article title which would be used to subsequently run the standard lookup.

4 Prototype Implementation

As mentioned earlier, we chose to organize our peers using the Chord structural overlay and to implement our wiki articles as Treedoc data structures. Lookups can be a major bottleneck in peer-to-peer systems so it is important to make them as fast as possible. Chord provides logarithmic lookup in the number of peers making it a reasonable choice as an overlay. Treedoc is great because articles can be stored and updated on any node by simply replaying logged operations. This feature allowed us to implement our highly-scalable caching mechanism.

Figure 3 presents a high-level view of our prototype. Essentially, peers interested in viewing and/or modifying an article pull it from the host peer, which is determined via a Chord lookup, to their local storage. Any changes to the article are logged locally. Once a client is satisfied with their changes, they push their log back to the host peer where it is replayed to incorporate the changes. Subsequent pulls, will see the updated article.

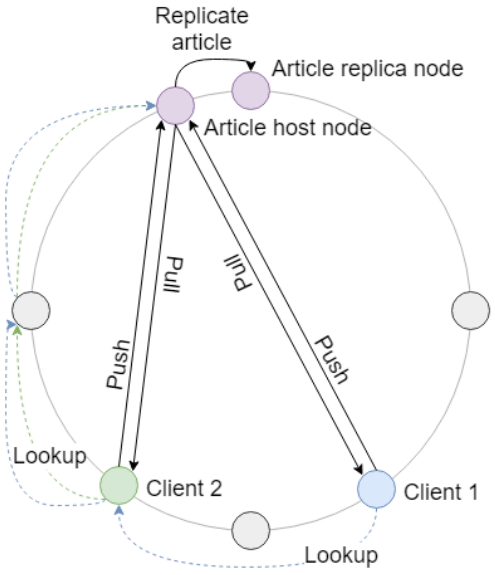

Figure 3: Describe figure.

The server daemon is run on each node participating in the Chord ring. It accepts lookup requests and serves articles to other peers in the network. It also manages the other Chord operations described in the Chord background section. All communication between servers is done using remote procedure calls (RPC) due to its natural semantics. It would also be possible to use a combination of protocols such as message passing and RPCs for different Chord operations. For example, lookups could use UDP and article transfers could use RPCs.

The client application is extremely simple, simply forwarding all operations to a server daemon using RPC. It is possible that a client could forward commands to a remote server daemon however, the expectation is that the client would connect to its local daemon since it would not know any other server addresses. In fact, this behaviour could have been enforced by using a different form of inter-process communication, for example pipes, but the RPC form allows clients to access articles without running a server. However, this may not be desired in a peer-to-peer network as it could introduce leechers.

To interact with an article, it must first be copied from the host peer to the client's local storage. The user may then view or edit the local version of the article; edit operations are logged on the client. The user can share their modifications by sending their log to the host where it is replayed on the shared article, or they can discard their local version without sending it. To support these operations, the client exposes five operations to the user: pull, view, insert, delete, discard and push.

- `Pull(title)`: This first checks to see if the specified article is already stored locally. If so, the command prints this and returns. Otherwise, it initiates a Chord lookup for the article. If the article is found, it is return by the lookup RPC and saved locally (in JSON format). Otherwise, a new article is created at the client.
- `View(title)`: Opens the local version of the article and prints it to the console.
- `Insert(title, position, text)`: Opens the local version of the article, inserts the given paragraph at the specified location in the article, logs the insertion path and the text, and saves it.
- `Delete(title, position)`: Like insert, except it marks the paragraph at the specified position as invisible.
- `Discard(title)`: Deletes the local copy of the article. Useful if a user wants to discard their local changes, or you want to pull an updated version of the article.

- `Push(title)`: Initiates a lookup for the article, sending the article's log with the lookup. Upon receiving the log, the host replays it and the edits are now available to any client's that make a subsequent pull.

5 Evaluation

6 Conclusion

References

[1] N. Preguica, J. M. Marques, M. Shapiro, and M. Letia. A commutative replicated data type for cooperative editing. In *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 395–403, June 2009.

[2] M. Shapiro and N. Preguica. Designing a commutative replicated data type. Report inria-00177693, INRIA, 2007.

[3] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *SIGCOMM Comput. Commun. Rev.*, 31(4):149–160, Aug. 2001.