YACET: Yet Another Collaborative Editing Tool

Yi Hui Chen
UC San Diego

Matthew Parker
UC San Diego

Thant Htoo Zaw
UC San Diego

Abstract

With the rising of software development, version control is introduced for better collaborations between developers. However, existing solutions have their flaws when editing the same file. Users need to resolve potential convoluted merge conflicts during the merge process or suffer from bloated application suites that are not suitable in a lightweight system. We introduce a lightweight, easily usable primitive for individuals to collaborate, YACET. YACET is built for collaboration in a remote file system, enabling users to define and lock inter-file ranges of data through Zookeeper [6] services to ensure integrity of the files. Through evaluation and testing, YACET successfully achieves its goal of enabling coordination between its users and demonstrates great potential.

1 Introduction

Collaboration between developers is becoming a trend along with the continuing rise of software industry. As such, new software and development workflows have gained popularity as means of facilitating collaborative work on projects. These are structured to handle the innate difficulties stemming from a distributed development model, including consistency, version control, merging, branching, and archiving history. Coordination between independent users is thus enhanced to allow for an easier and more efficient development process.

Arguably the most famous of these is Git [1], an open-source distributed version control system that is quickly becoming the industry norm. It provides all of the aforementioned features in a lightweight, streamlined model in which the entire source tree is replicated at each end for usability. Large alterations to a project are written locally and pushed all at once. Algorithms are employed to merge commits if feasible, but conflicts frequently arise that must be manually corrected, especially as contribution activity heats up. Therefore, a sizable tolerance for stale data and patience for actively resolving conflicts is advised.

In a different avenue, collaborative editors such as Google Docs [4] typically involve a single master repository on the server with clients making incremental changes remotely. This necessitates a focus on low latency to create a "real-time" experience in which changes are propagated to all clients almost immediately. Although consistency is maximized, the requirement of centralized storage only on the server is a limitation that makes this model inappropriate for some contexts.

Systems have supported forms of intra-file locking for quite some time. The concept goes by the name of "record locks." [5] It is most typically associated with databases, whose fixed-sized table-of-records structure gives way to an easy, organized, consistent procedure for locking small sets of fine-grained data in the context of a much larger collection. MySQL [2], for example, offers read/write record locking capabilities during a SELECT query. File systems also support forms of record locking. Unix has process-associated record locks standardized by POSIX that may be obtained through the *fcntl* system call.

It was the file system fine-grained locking features (or rather, lack thereof) that inspired this project. We introduce YACET, a prototype command-line tool that ensembles and maintains a structure for versatile locking of data for collaborative work. It serves as a starting point to build off of traditional fine-grained locking strategies. These are often satisfactory for most purposes, but leave plenty of room for expansion in the following ways:

- **Inter-file lock ranges**: Record locks are limited to a simple one-time base-and-bounds extent of data within a file. YACET's augmentation allows for arbitrary *interfile* locking of related data.
- **Named ranges**: Record locking is typically done in an ad-hoc manner; acquiring a region or a group of regions must be done in an iterative fashion. YACET stores persistent ranges consisting of a set of intervals spanning potentially many files that automatically adjust to edits and can be looked up by name.
- **Secure locking**: Both Unix and MySQL record locks are advisory, meaning involved parties must cooperate to ensure data integrity. Mandatory locking is non-POSIX, yet is supported on Linux at a cost of potential race conditions with read/write calls. The design of YACET keeps fragile file edits intact without races, and the program as a whole can be made "mandatory" through a combination of setuid and designating a sticky directory.

Locking is implemented with MySQL and Zookeeper, which work in tandem to guard file intervals as they are in use and to synchronize data transfer between users and the backing storage.

2 Motivation

In YACET, we focus on supporting fine-grained locking of arbitrary selections of data. Our initial motivating example (in fact from personal work experience) is the procedure of updating JSON files that store product parameters and settings. By its nature, the data was categorized into a large number of distributed files yet still shared some relative structuring and values. Executing a change request required either a (blocking) checkout of a sect of the repository or enough luck that the update was transient to not be disturbed. The former caused an unacceptable bottleneck, as the data was so massive and widespread, so chances had to be taken with the latter.

A fair argument can be made that separate files with overlapping or otherwise interdependent data should be coalesced. This is not always the case. A product line that branches may require its own configurations yet still have plenty in common with its sibling. Even within the same product, there may be multiple profiles that each need their own files. Also, having some form of redundancy in some cases allows for performance gains because unneeded data can be scrapped.

The structure of JSON makes the solution to this problem even more straightforward. For each atomic entity (e.g. a JSON object), an interval may be specified that encapsulates it in much the same way as the curly braces of the syntax. A modular range may be created that houses all relevant entities spanning any number of files. Then, exclusion is enacted according to this range and nothing beyond, and the range is automatically adjusted to reflect changes both within it and from other ranges. This is the high-level essence behind YACET.

2.1 The Scope of This Project

We first emphasize the infancy of YACET as an exploratory proof-of-concept program built into a prototype Linux command line tool. As such, some aspects of it are provided as unpolished “black-box" implementations. These parts of the tool, albeit essential for correctness, are simply not our focus for this project.

- A MySQL database is our choice for the backing store of metadata that describes and gives structure to the data that is part of our fine-grained locking scheme. The backing store may very well be an abstraction that simply provides this capability.
- We implement our distributed locking service using Zookeeper, since it provides a replicated persistent storage and automatic sequence numbers useful for locks.
- Since YACET is experimental as of now, we loosen security constraints regarding accessibility of the file system, MySQL database, and Zookeeper.
- YACET does not have a lavish variety of luxury option flags as many ubiquitous Linux user binaries do.
- We developed and tested YACET on NFS [7], although its requirements do not extend beyond the POSIX standard. YACET works on a local file system as well.

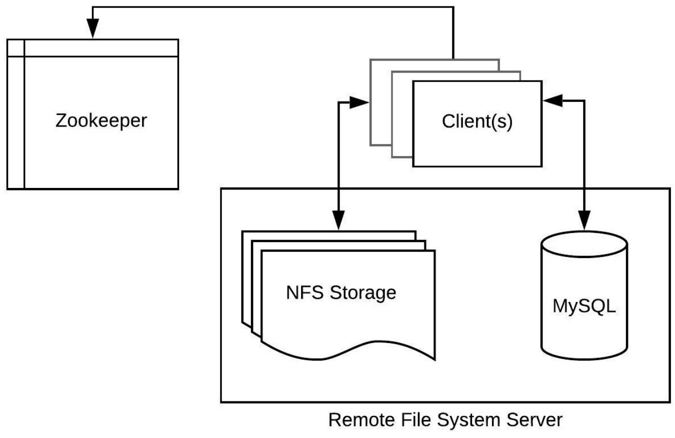

Figure 1: High level design of modules - the client talks to zookeeper to obtain locks, queries the relational database for named ranges and communicates with NFS for the files.

Indeed, the last point is of particular noteworthiness, as YACET was originally conceived to tie in with the file system. Instead, it is presently a preliminary exploration into the possibility of integrating special fine-grained locking features into a networked file system. As such, the features outlined in this paper are guaranteed only if the files are accessed via our command line interface.

Additionally, our main drive is to renovate the manual labor of coding in industry. Therefore, while theoretically applicable after the above grievances are strengthened, YACET is not designed for integration in high-performance data centers that conduct automated high-bandwidth writes. Rather, its use is intended for comparatively leisurely manual file edits by a user at a workstation.

3 High Level Design

YACET enables concurrent editing of shared files over the network via a user level client command line program, hereby referred as “client". Our client acts as the entry point for the user to access the files. Each user can specify ranges, which we define as a group of regions over one or more files, that they want to work on, which our client will open up with the specified executable editor. The requested files live on a shared remote file system, for which we use NFS for its ubiquity. The information about ranges are stored in a relational database, for which we are using MySQL. This database lives on the same server as NFS, since our initial prototype only runs on one remote file system and each file’s range information becomes meaningless once the NFS storing the actual files goes away, without assuming any kind of replication on the file system. To support concurrent clients, we utilize a lightweight MySQL record-locking scheme for atomic database updates and implement a distributed locking service using an ensemble of Zookeeper servers, which provides two sets of locks to the client, “interval" and “master" locks (see Sections 4.3.1 and 4.3.2). As shown in Figure 1, the different components of the system communicate via the network, and the client has built-in retry logic for minimal network failure.

4 Implementation Details

4.1 Client Interface

Clients access YACET’s capabilities through a preliminary command-line interface user program. A client may use the tool in any permissive working directory, and at the onset the set of files to edit extends to the reachable file system. Additional organization, such as specifying a directory only under which files may be edited, can be achieved on top of YACET via sticky and setuid permission bits.

The backing store houses named ranges that are referenced by name from the command line. Users are met with the following modes of usage:

- -w: Write mode. A user specifies a named range and a set of files within the range to open. Changes made within the range’s intervals in the file will be persisted.
- -r: Read mode. Operates the same as write mode, but no changes will be persisted.
- -g: Create mode. A user specifies a named range, a set of file paths, and for each file path a set of byte offset pairs that indicate the inclusive start and exclusive end of an interval in that file. The range is constructed and save in the MySQL database.
- -p: Print mode. A user specifies a list of range names and file paths. For each range name, all files and intervals of the range are printed to the terminal. For file paths, all intervals associated with that file are printed.

For read and write modes, the user must also specify an executable (which we will refer to as “the editor" from now on) with which the files will be opened, along with any additional flags to be passed into the editor. This introduces the first obstacle in the design of YACET: we aim to allow a proprietary program of the user’s choice to edit the files. However, such programs have no knowledge of the range information behind the file, so the user isn’t restricted to the designated intervals of the range. In order to impose these controls, we duplicate the backing file, creating what we refer to as “our swap file". This also grants the added benefit of easing the merging of writes into the backing file, similar in faith to how many text editors use their own swap file. In their case when used in conjunction with YACET, this would actually be layered as a swap file of our swap file of the backing file.

With this technique, we can explicitly control what the user reads and writes. One option is to only send the relevant data within the intervals, but we understand having the other parts of the file for reference is also important. Therefore, YACET sends the entire file to the editor, but beforehand, it inserts “oracles", small predefined strings, to notate where the intervals begin and end. These strings can be set by full path name, file name (without path), and/or file extension in a file named “~/.oraclerc".

Edits may easily be checked for obedience in our swap file before finalizing the writes to the backing files. YACET handles two cases of improper edits that may occur:

- The user edits outside of the oracles. YACET only takes bytes between the oracles, so these changes are disregarded anyway. If bytes before the oracle are edited, then the offset of both oracles could potentially shift up or down. To solve this, YACET saves the offset of the opening oracle before running the editor so it always knows where the interval *should* be.
- The user edits the oracles themselves. YACET knows how many sets of oracles there are, so it will refuse to accept any changes if it cannot find the correct number of intact oracles. Rather, when this case is detected, an error is printed and the editor is relaunched to allow the user to fix the oracles.

If YACET determines the edits are satisfactory, the updates are then propagated to the MySQL database, which we describe next.

4.2 Range Store

The information used by the client program is stored in MySQL database on the server. All clients have equal privilege accessing the database. The schema of the database is shown in Figure 3. Parallel C structs follow closely with the tables and relationships in the database.

There are two primitive tables in the schema, they are the “File" table and the “RangeName" table. The former stores file paths, and the latter stores the name of the user defined Range and its initialization status.

The last two tables are the Offset table and the RangeFileJunction table. These connect the two primitive tables and ensure the core functionality of YACET. The Offset table stores the starting (base) and ending (bound) byte offsets of each interval associated with a given Range and File. Files are not replicated for each range because keeping a one-to-one mapping of File table records to actual files on NFS is essential for the Zookeeper service to function properly. Instead, the RangeFileJunction table is used to create a many-to-many relationship between ranges and files. This permits the ability to store both inter-file and overlapping ranges.

YACET contains four procedures for querying the database, each wrapped in a transaction.

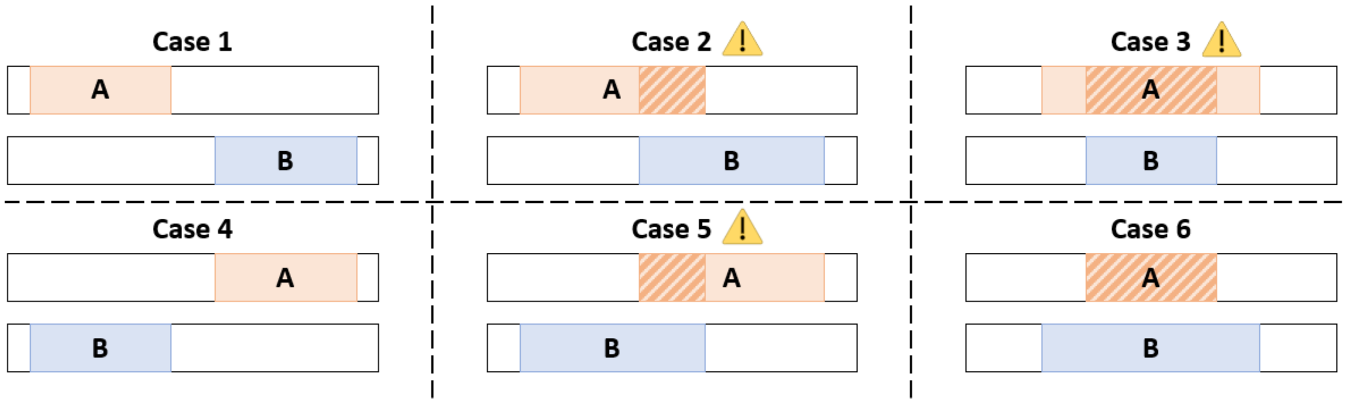

Figure 2: Interval shifting cases. Problematic cases have a warning logo.

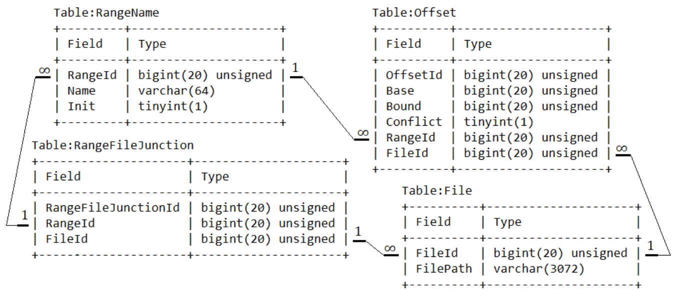

Table:RangeName

Field	Type
RangeId	bigint(20) unsigned
Name	varchar(64)
Init	tinyint(1)

Table:RangeFileJunction

Field	Type
RangeFileJunctionId	bigint(20) unsigned
RangeId	bigint(20) unsigned
FileId	bigint(20) unsigned

Table:Offset

Field	Type
OffsetId	bigint(20) unsigned
Base	bigint(20) unsigned
Bound	bigint(20) unsigned
Conflict	tinyint(1)
RangeId	bigint(20) unsigned
FileId	bigint(20) unsigned

Table:File

Field	Type
FileId	bigint(20) unsigned
FilePath	varchar(3072)

Figure 3: Database Schema

1. **SelectNamedRange**: Provided a range name, select the files and offsets that make up that range. A read record lock is acquired while the intervals are checked to make sure no other user as open a conflicting range.

2. **SelectFileIntervals**: Provided a file path, select all intervals in the file. This is mainly used by Zookeeper, as described in Section 4.3.1. Additionally, it is used for printing files in print mode.

3. **InsertNamedRange**: From the command line parameters given in create mode, insert a new named range containing these parameters. The range name is first added to both verify it is unique and to set the “Init" field to false. Once the rest of the data is inserted, this field is set to true to indicate the range is ready for use.

4. **ResizeFile**: Given a file and its intervals updated from a file write, apply these changes to all affected intervals in the database. Since the input intervals themselves might have been shifted in the duration of the user’s edit, current values are first fetched within a write record lock. Since undoing a file write is more difficult than undoing database changes thanks to the ability to roll back transactions, the querying is done to completion before the writing to the backing file, and finally the record locks are released.

4.2.1 Shifting Intervals

The standout feature of YACET is the ability to edit intervals of two different ranges in the same file at once. The issue is, when the size of the earlier interval changes, the offsets of the later interval are affected as well. For how interval A’s alteration affects interval B, there are six cases to consider, depicted by Figure 2 (δ_A is the change in the size of A, i.e. $(\text{bound}_{Af} - \text{base}_{Af}) - (\text{bound}_{A0} - \text{base}_{A0})$).

1. B is entirely after A; it is shifted by δ_A.

2. Part of B is after A and part of A is before B; B’s offset is shifted by δA, but its base may exist anywhere within A.

3. B is entirely encapsulated by A; B’s base and bound may exist anywhere within A.

4. B is entirely before A; it is unaffected.

5. Part of B is before A and part of A is after B; B’s base is unaffected, but its offset may exist anywhere within A.

6. B entirely encapsulates B; B’s base is unchanged but its bound is shifted by δ_A.

Cases 2, 3, and 5 are unfortunately detrimental: since we are only aware of the new and old base and bound offsets, there is no way of knowing how some components of B are affected. In Figure 2, changes to the striped regions of A would change the size of B, while other changes would not. Cases 2, 3, and 5 are problematic in that A has a mixture of striped and non-striped parts. Case 3 is especially devastating: a user can remove all bytes of $A \setminus B$ and add the same number of bytes within $A \cap B$, effectively expanding B to match A exactly, regardless of how small B began.

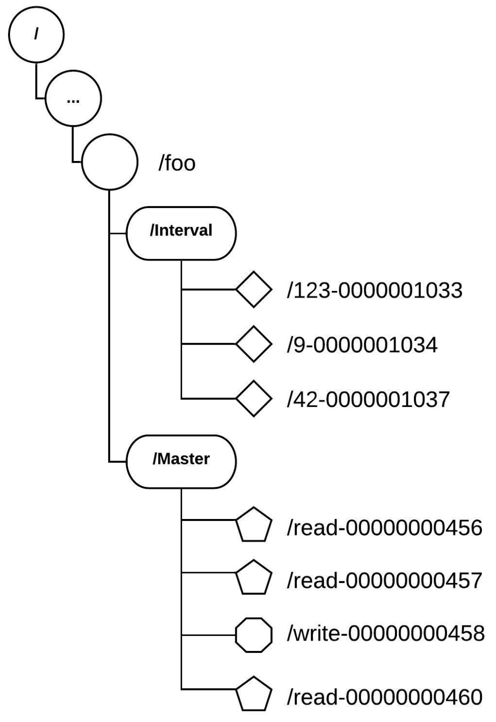

Figure 4: Lock structure for file "foo" in Zookeeper

We leave all offset values within the striped sections of an altered range untouched, and instead set the “Conflict" boolean of the offset entry in the database to true, which warns the user of a potentially unreliable interval. This value is falsified when the interval itself is edited. A possible strategy for dealing with this issue is to fetch all intervals that conflict with a range and insert oracles for them as well. We decided this would be too much to keep track of at this stage, plus the user might eventually have to contend with and keep intact a mess of oracles scattered within the editor. Practically constructed ranges would rarely overlap with each other anyway, as such intervals try to limit their reach to only what is necessary, thus are likely atomic.

4.3 Distributed Lock Service

We set up an ensemble of three Zookeeper servers for use as a distributed lock service. Our client program maintains a Zookeeper client session with the Zookeeper servers. All the locks concerning a file are placed in the directory of that file in zookeeper. The top level has directories for each of the NFS servers.

We introduce two types of locks: interval and master locks. Both are created using the *Ephemeral* flag to ensure that the lock znodes get deleted upon client disconnect, and the *Sequential* flag to obtain a consistent ordering of the znodes being created concurrently. For all the locks, the *Sequence* number gets automatically appended by Zookeeper as a zero-padded ten-digit 4-byte integer after the “-" character. Since the *Sequence* is maintained by the parent directory, we chose to split the locks into 2 different sub-directories: “/Interval" and “/Master" for separation of concerns, as well as ease of filtering and sorting.

4.3.1 Interval Locks

The interval locks are used to indicate back to the user if a particular region was being modified. Any individual region of a file being requested by the user, identified by its unique primary key *OffsetId* in the **Offset** table, corresponds to an interval lock in Zookeeper with the structure of <OffsetId>-<Sequence>, where the *Sequence* is a monotonically increasing number maintained on the “/Interval" directory znode. The process for acquiring an interval lock is as follows:

1. The znodes representing the file path in NFS is created recursively in Zookeeper.

2. An interval lock is created with the requested region’s *OffsetId* in the database in the “/Interval" subdirectory of the actual file path in Zookeeper.

3. The client gets all the children of the *Interval* znode to obtain all existing interval lock requests, and sorts them by *OffsetId*. This sorted list of interval locks is primarily used for cross-referencing with the *OffsetIds* obtained from the database in the Step 4. An ideal solution would be a map with *OffsetId* as the key and the list of *Sequences* corresponding to the multiple lock requests for the same region, but this was foregone for ease of implementation.

4. The client queries the database for all regions in the file, and processes the results to obtain the list of regions in the file conflicting with the requested region.

5. The list of conflicting regions is then cross-referenced with the list of interval locks by using binary search, to create pairs of <*OffsetId*, *Sequence*>. These conflict pairs are then sorted by *Sequence* in ascending order.

6. If the requested region’s *OffsetId* from Step 1 is at the start of the lock, indicating that it holds the smallest *Sequence*, the client has successfully acquired the lock.

7. If not, the client returns a failure, indicating an ongoing concurrent conflict.

Releasing the interval lock towards the end is as simple as deleting the znode made when the interval lock was first acquired, using the actual file path (including the *Sequence* number) returned by Zookeeper.

An alternative future implementation where the interval lock makes use of Zookeeper watchers to form an interval lock queue will be explored in the Section 7. In our current implementation, if the client detects a subset of the user's requested regions being held up in interval locks, the client releases all the new interval lock requests and tells the user about a concurrent update blocking their edit request at the current time. We picked this approach over blocking the client and queuing up in the order of locking for the following reasons:

- **User Experience**: We consider it a poor user experience if the client were to block the user's terminal interface the whole time the interval locks are being queued up.
- **Deadlock Prevention**: Because our interval lock implementation is tightly coupled with regions stored in the database and the user requests are in terms of ranges that cover multiple regions, the client has to grab an individual interval lock for each region. For instance, given two clients: client 1 requesting regions A, B, C, and client 2 requesting regions C, D, A, when processed in order, they could end up blocking each other's locks as client 2 holds up the interval lock for region C that client 1 needs, while client 1 holds up the interval lock for region A that client 2 needs. This is prevented in our current approach by enforcing an all-or-none mandate on the interval locks, by simply choosing not to hold up all interval locks if any one of them is unavailable.

4.3.2 Master Locks

In contrast to interval locks, master locks are multiple-reader, single-writer locks on a per-file granularity, created under the "/Master" subdirectory of the file path in Zookeeper. Our implementation is adapted from Zookeeper's recipe for shared locks [3]. The master locks are utilized any time the client interacts with the backing files in the remote NFS, in these two situations:

- **File Open**: In order to service user requests, upon acquiring the relevant interval lock(s), the client makes a copy of the requested file in a local swap file. This also involves querying the database about the region being requested. In order to ensure the atomicity of these operations and serializability of reads against writes, we make the client request a master read lock:
    1. The client creates a znode with the prefix "read-", with the automatically appended sequence number, which we will call *currentSequence*.
    2. The client then obtains the list of all children of the *Master* znode, filters only the znodes with the "write-" prefix and searches for the lowest sequence number *minWriteSequence*.
    3. If there is no *minWriteSequence* lower than *currentSequence* of the requested master read lock, the client has successfully acquired the master read lock.
    4. Otherwise, the client sets a watcher on a master write lock znode with the highest sequence number that is lower than *currentSequence*, the sequence of the requested read lock's znode.
    5. In the meantime, the client's execution is blocked using a *pthread_mutex*, since this read lock is merely waiting for some finite amount of writes and reads to complete. When the watch event gets triggered by the write lock znode being deleted, the client will repeat the steps from Step 2 to reattempt acquiring the read lock.

  The master read lock is held only for the duration in which the client makes the copy of the backing file in NFS to a local swap copy. Effectively, the master read locks wait until all writes with lower sequence numbers are serviced.

- **File Close**: At the end of user edits, the editor executable gets closed and the changes get written to our local swap copy made during the file open. The client then updates the ranges in the database for the file should there be any changes in the current interval's size, as detailed in Section 4.2.1. The write process to NFS is then initiated. First, a more recent version of the backing file is copied over. The changes in the local swap copy are then updated in the relevant sections marked by our oracles (detailed in Section 4.1). This version is then pushed to NFS, where it replaces the backing file. All of these operations are guarded with a master write lock to make them atomic and to guarantee the backing file's state is not disrupted throughout the write procedure. To obtain a master write lock:
    1. The client creates a znode with the prefix "write-", with the automatically appended sequence number, which we will call *currentSequence*.
    2. The client then obtains the list of all children of the *Master* znode, and searches for the lowest sequence number *minSequence*.
    3. If there is no *minSequence* lower than *currentSequence* of the requested master read lock, i.e. there are no read or write locks with a lower sequence number, the client has successfully acquired the master write lock.
    4. Otherwise, the client sets a watcher on a master read or write lock znode with the highest sequence number that is lower than *currentSequence*, the sequence of the requested write lock's znode.

5. In the meantime, the client's execution is similarly blocked using a *pthread_mutex*. When the watch event gets triggered by the watched znode being deleted, the client will repeat the steps from Step 2 to reattempt acquiring the write lock.

Similar to the master read lock, the master write lock is only held for the duration of the actual file save. The master write locks ensure there is only one single writer at a time, and then reads are ordered between writes.

4.4 Full Procedure

With the three main components outlined, we now piece together the full flow of a YACET write operation. Figure 5 illustrates the trace. The yellow blocks signify database queries, the green represent interaction with the Zookeeper service, and the grey indicate the client program's interactions with the user and the file systems - both NFS for the remote backing file and the local file system for swap files.

Since ranges can involve multiple regions spanning different files, we make use of POSIX threads to parallelize procedures for each file. After successfully acquiring all the required interval locks, YACET splits off the processing of files into threads, which each pull its respective NFS backing file into the local swap file while holding a master read lock and adding oracles. At this point, one of two routes may be taken depending on the command line flag for defining th executable:

- -e: Each thread forks the editor in a new process. No thread synchronization is performed. This is useful for editors that open an external window. Upon closing, the thread does not wait for others' editors to close.

- -E: The threads rendezvous after pulling the files, and the main thread collects all file paths and forks the editor in a single new process. This is useful for terminal-based editors like Vim, for which the command "vim file1 file2 ... -p" may be executed to display multiple files in one vim process at once.

When the user's edits are done, the threads resume, first verifying integrity of the edits. For either option, only files with corrupted oracles are relaunched in the editor; other threads carry on independently, performing interval shifting queries on the database, and pushing the changes to NFS.

5 Evaluation

Due to the nature of YACET, it is hard to measure performance garnered while using the application versus the normal file editing experience: the main benefit would be the time difference between concurrent and sequential edits, which is

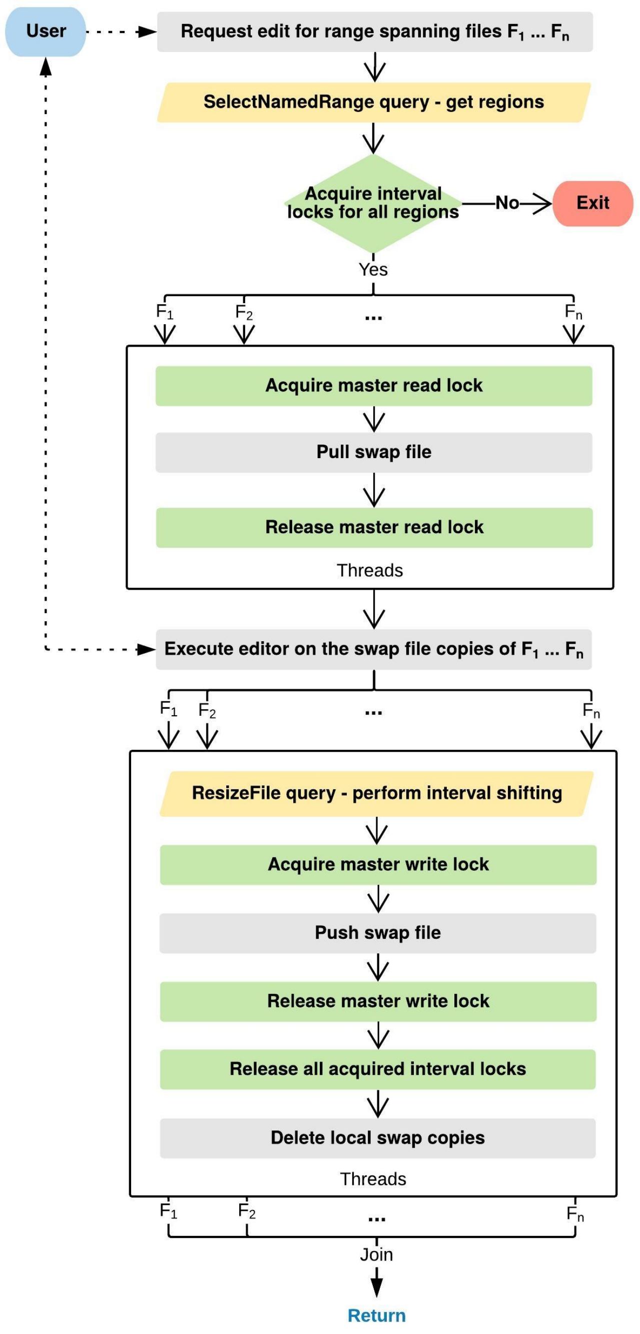
User
Request edit for range spanning files F_1 ... F_n
SelectNamedRange query - get regions
Acquire interval locks for all regions
No
Exit
Yes
F_1
F_2
...
F_n
Acquire master read lock
Pull swap file
Release master read lock
Threads
Execute editor on the swap file copies of F_1 ... F_n
ResizeFile query - perform interval shifting
Acquire master write lock
Push swap file
Release master write lock
Release all acquired interval locks
Delete local swap copies
Threads
Join
Return

Figure 5: End to end procedure of a user editing a range spanning across multiple files. NFS/local file system accesses are in gray, MySQL accesses are in yellow, and Zookeeper accesses are in green.

highly dependent on the user and the job. Instead, we present the evaluation on the correctness of our system and the performance overhead introduced.

The experiment setup we have emulates that of Figure 1 where multiple client instances connect to a single NFS and MySQL server along with three zookeeper servers, with the exception of multi-client evaluation, which will be discussed in section 5.2.

5.1 Concurrency Correctness

To validate the correctness of our implementation, we tested several scenarios and verified the corresponding output in the database and the files in NFS.

We first created a new range with the -g option, supplying a file path and interval offsets to the program. After the application successfully returned, we verified that these inputs matched those in the database. We then attempted to create a new range with the same range name from another client. Because range name is unique, this operation correctly returned an error to the application.

Following that, we then tested three scenarios users might encounter when collaborating using the application, all involving multiple users accessing the same file.

1. There are no overlaps in the ranges users are trying to access.
2. Some portion of the ranges overlaps.
3. Multiple users are using the same range.

In scenario 1, because there were no conflicts between the users, both users were able to proceed as if they had exclusive access to the file. Both content changes were saved regardless the order of termination. In scenario 2, an error occurred when the second user attempted to access the file. This is because the region declared by the first user became protected, resulting in termination of the application for the second user. The same reasoning can be applied to scenario 3, but in this example the second user accessed a range that was directly in use. We also tested the cases from section 4.2.1, and intervals stored in the database were shifted accordingly.

5.2 Performance Overhead

To measure the performance overhead introduced by our application, we separated the measurement into two parts: creating a range and accessing the file. The overheads include database contention, zookeeper lock management, and searching through the intervals. The following measurements were done on a single client.

Because it is unlikely for the user to have a large number of files mapping to the same range, we limited our measurement to five files. For range creation, the overhead was measured by timing the starting and the ending of the application. The result shows that overhead increases with the number of files the user tries to link to the range. However, this increase is also dependent on the state of the machine. As we can see in

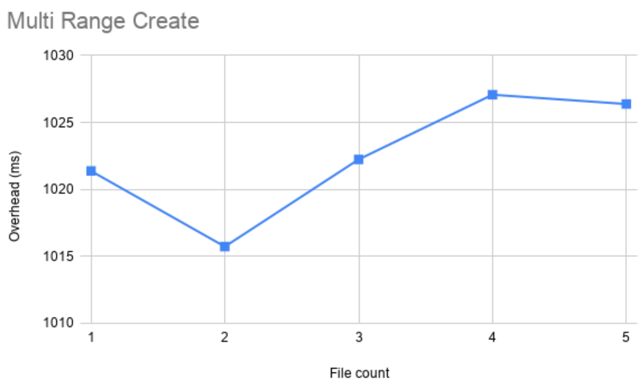

Figure 6: Performance overhead of creating range that contains multiple files

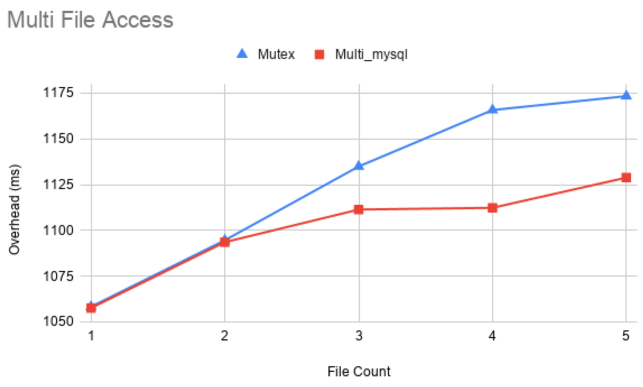

Figure 7: Performance overhead of opening multiple files in the same range

Figure 6, there are slight increases in the overhead but also sudden drops. We concluded that this is most likely explained by operating system scheduling.

File access (writes) were measured by setting up two separate timers: one before entering into the text editor and another right after, as the time it takes to edit the files is not part of our application. The result in Figure 7 shows that there is a steady increase in latency with the number of files the user tries to access. For this test, we collected two sets of data, each for a different implementation: serializing writing to the database on one connection, or spawning multiple SQL connections in each file thread. This is necessary because the SQL library does not support multi-threading on a single connection instance (further discussion in section 6.2). The results show that for many files, thread spawning outperforms sequential commits, and it therefore worth the potential overhead of instantiating a new MySQL connection object.

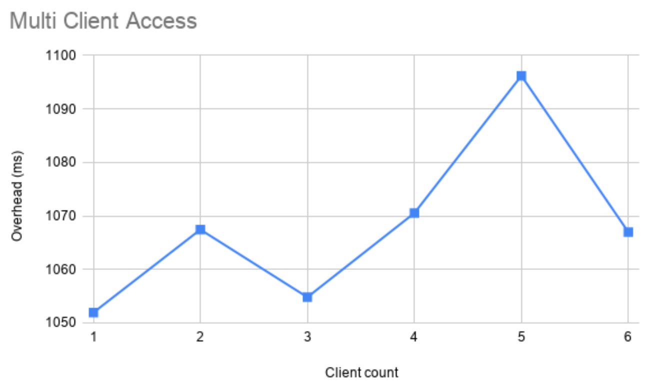

Figure 8: Performance over head when multiple client trying to access the same file with different ranges

Lastly, we attempted to measure the overhead when multiple users try to access the same file with the ranges defined to be non-overlapping. This is done by spawning multiple client sessions on a single client machine, collecting timing in the same manner as the file access test, and passing *cat* as the executable. Such decision were made due to difficulties in synchronizing the actions of starting the client and exiting the editor on different machines.

We expected some overhead as there were to be multiple access to the database and zookeeper systems. The result (Figure 8) is not as promising because there are no obvious patterns with the number of clients. This may be due to the measuring technique, but Without the knowledge of how to sync up multiple machines, we fail to come up with a better strategy.

6 Alternative Design Decisions

The following section discusses some of the decisions we made and the reason behind it.

6.1 Operational Transformation

Section 4.2.1 detailed our approach to Operational Transformation by relying on record locks in MySQL and applying the interval shifting for a file's regions on every write of the file. An alternative approach that we considered along the way involved versioning the file and logs, as well as storing all the relevant interval information and the current file version in Zookeeper without coupling the database. All interval locks would store the file's version and byte offsets at which the lock node was created. Each write to the file, guarded with master write locks described in Section 4.3.2 for atomicity, would:

1. first get the current file version, and write a znode with the current version, the file's old bound and the size change delta to the "/Logs" sub-directory of the file without any flags (not *Ephemeral* since we want the log node to persist, and not *Sequential* because we order it using the file's version), and
2. increment the file version, which was stored using Zookeeper's verison metadata on a single znode and updated with *setData()* calls with empty data every write.

On every read of the file, guarded with corresponding master read locks for atomicity as well, the client would:

1. check the file's version on the interval lock against the file's current version to determine a potential need for interval shifting,
2. and if needed, would go through all the logs, filtering to those above the interval lock's file version, and applying the size deltas if any of the offsets is greater than the bound stored in the log.

This approach offloads the work of shifting all relevant intervals for the current file on every write, as done in the current approach, towards doing this on every read. However, we abandoned this approach due to these drawbacks:

- While using file versions and logs enables a potentially higher write throughput due to not having to block and update the relevant intervals on every write, it is not suitable for read loads, which are also part of every write. This translates into a bad user experience waiting for the file to load up, due to having to catch up on shifting intervals while filtering through all the logs in the file's history.
- Since we were already storing information for named ranges in the database, storing interval data in Zookeeper unnecessarily duplicates data storage and introduces extra delays as a side effect to the database due to queries inside file open and close operations which have to be guarded with locks for atomicity.
- Lastly, since our approach involves putting all the logic inside the client interface, it was hard to figure out a consistent way to garbage collect, which would be necessary due to Zookeeper's nature of not being intended for storage purposes.

6.2 Commit Methods

As mentioned in section 5.2, after the client program finishes editing multiple files, there was a problem committing the change. In particular this issue occurred in the MySQL library. Our implementation spawns one thread per file (described in Section 4.4), each of which pushes its file change to the NFS server. These threads would then commit the metadata change using the same MySQL reference, which resulted in a race condition that corrupted the MySQL reference.

Initially, our fix was serializing metadata writes to the database, but we quickly developed a second solution to spawn a MySQL references for each file writing thread, as the NFS writing procedure is already multi threaded, enabling parallel database access.

It was unclear which implementation is the most optimal at first. For the first solution, the bottle neck would be the push to database and this overhead scales linearly with the number of files. On the other hand, the program will require more memory usage and can slow down the database write by overload it with requests. Through the measurement in Figure 8, the results show that multi-threading the request has the least database contention and is more favorable.

7 Future Work

What we present in this paper establishes the potential use cases of concurrent collaboration for semantically separate but coupled pieces of information in files. While our prototype serves the purpose of showing the feasibility of using locks around regions in the form of named regions, support for other ways of separating information in files is a goal for the near future. This can involve:

- raw interval ranges without having to create ranges
- line number ranges
- regex pattern matching (such as locking a code block between braces)

One of our original motivations behind YACET was to allow inserts into any offset as long as the offset is not in a region held up by an interval lock. While our current write option allows pseudo-inserts in the form of a write request for a region that spans just 1 byte, a proper insert could potentially prove useful to an append heavy workload, where multiple users can append to the end of the file at the same time without being blocked.

Moreover, while we explained our decision to reject any edit requests that conflict in Section 4.3.1, a more mature version of YACET could add the option to ask if the user desires to block and wait for interval locks, implemented using sequence numbers in Zookeeper to emulate a lock queue. To prevent deadlocks, possible venues of exploration might involve making the process of acquiring all interval locks atomic, or introducing a different type of range lock rather than the file interval locks that would span across multiple files.

We also chose to use NFS for its ubiquity as a remote file system that can be mounted and accessed from multiple clients. A future version could be made portable for other distributed file systems so that data does not get lost on storage node failures, and make the entire workflow fault tolerant, rather than just the distributed Zookeeper service. With a more polished version of YACET, it would be an interesting evaluation from a user experience standpoint of our tool to beta-test with multiple users in different kinds of workflows and survey them to discover which kinds of use cases YACET fulfilled better than their usual collaborative workflows and to hone in on those for feature development.

8 Conclusion

With the rising in software developer collaborations, allowing multiple users to modify the same file significantly increases the productivity. However existing version control software and collaborative editing tool all have their weaknesses. In this paper we propose YACET, a file system tool that allows users to define the portion of the file they are working on and enables them to modify its content simultaneously. Due to time constraint there were a lot of features that we were not able to implement. However, through evaluation and testing, the result of YACET is promising. Although YACET is most beneficial when working on shared files in the file system, it can also be used by user to prevent accidentally deletion of file content. These use cases can grow as the application becomes more refined.

References

[1] Software Freedom Conservancy. Git - about version control.

[2] Oracle Corporation. Mysql.

[3] The Apache Software Foundation. Zookeeper recipes and solutions, 2020.

[4] Google. Google docs.

[5] Jim Gray and Andreas Reuter. Distributed transaction processing: Concepts and techniques, 1993.

[6] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. *USENIX annual technical conference*, 8, 2010. https://static.usenix.org/event/atc10/tech/full_papers/Hunt.pdf.

[7] R. Sandberg. The sun network filesystem: Design, implementation and experience, 2001.