Tra - A file synchronizer

CS244b Fall 2017
Jennifer, Chih Wen, Lin
Kim Truong
Hemanth Kini

Overview

Tra is a file synchronization program. It provides a simple yet efficient algorithm to perform file synchronization across an arbitrary number of replicas. It uses vector time pairs to correctly synchronize any pair of replicas, seamlessly noting conflicts wherever they occur. Tra can track changes to files and propagate those changes from replica to replica, using network bandwidth that is proportional to the size of the changed files. In this project, we implemented a command-line version of Tra with Python, using gRPC as our communication framework. The implementation is designed to minimize the overhead of setting up any replicated file system. The user can simply run a command line operation to sync any directories to another device as long as there is a Tra server running on the device. In addition, we integrated an algorithm similar to rsync to achieve lower network bandwidth when syncing small changes for large files.

Goals

Tra has the following 5 goals that we were able to achieve with our Tra implementation:

1. *No synchronization order among replicas*: We tested this by synchronizing among 3 replicas in an arbitrary order and verified that the results were consistent.
2. *Conflict Detection Without False Positives*: Conflicts can be falsely detected in such scenarios as Figure 17 from the paper[1] using synchronization tools like Unison. We have verified that our implementation does not falsely detect conflicts that scenario.
3. *Zero Storage for File Deletion Metadata*: We detect whether a file is new or has been deleted with no extra storage costs by adding deletion notices in the parent metadata.
4. *Bandwidth Consumption Proportional to Syncing Set*
5. *Partial Synchronization*: Goals 4 and 5 are related and achieved in our Tra implementation by returning “Do Nothing” to the client when a directory is already up-to-date. The client expends bandwidth only on partially synchronized directories.

Design

Implementation

Our implementation of Tra is designed to be run as a simple command line tool without setup overhead. A sync command can be run from any given directory to any replica as long as the replica has a Tra server instance running. Tra only creates metadata to store sync state when receiving a sync command (or loads previous metadata from disk using a filesystem module.) It uses gRPC to handle serialization and network connections. This provides a seamless way for any two Tra instances to communicate with each other - the user only needs to provide the hostname and (optionally) a port, and gRPC will handle setting up the connection,

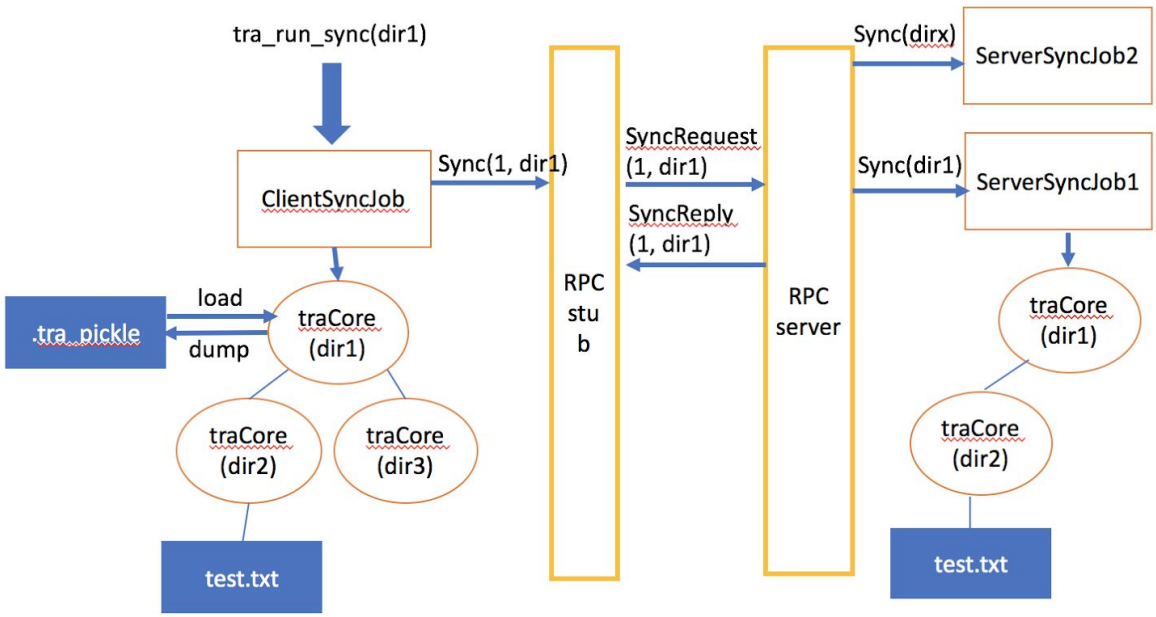
serializing and deserializing data, and calling appropriate stub routines. We use streaming connections to handle uploads in order to improve performance.

Synchronization

The following diagram shows the control flow for a sync process. A `clientSyncJob` is created when the program is started with a given directory to sync. The job creates or loads `traCore`s for each subdirectory and updates it against the file system. It recurses down the file tree in a BFS pattern, sending over its vector time pairs through a sync RPC call for each directory. If the corresponding `traCore` on the server replies COPY, it sends the vector time pair for its files through another sync RPC and keeps recursing down the file tree. Else, if a DO_NOTHING is received, the recursion will terminate. This allows us to achieve goals 4 and 5 by not wasting bandwidth on already up-to-date directories.

On the server side, a `serverSyncJob` will be created with a job id when the RPC server receives a new `syncRequest`. The RPC server can theoretically serve multiple clients' requests in the same time, as it loads separate requests in separate jobs. The `serverSyncJob` will load or create the `traCore` for requested directories. The job can detect missing files based on the assumption that the `clientSyncJob` will walk down the file tree in a BFS pattern.

Figure 1: Tra Architecture Diagram



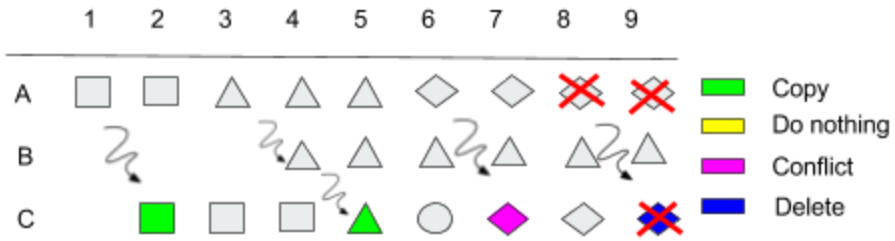
Vector Time Pair Syncing

Tra is designed around the concept of *vector time pairs*, which is comprised of a modification and synchronization vector. Vector consists of dictionary entries { replica ID : timestamp }. A modification from a certain replica would record an entry (or update it if the replica's entry already exists) with the replica's ID followed by the timestamp at the time of the

modification. Similarly, a synchronization to a certain replica would record an entry with the replica's ID followed by the timestamp at the time of synchronization. The synchronization vector is a superset of the modification vector with the extra synchronization information because the modification vector by itself is insufficient to detect conflict without false positives.

According to the paper's Figure 17, the following scenario in Figure 2 below with 3 replicas A, B, C where different shapes denote different versions of the file and a squiggly arrows (~>) denote unidirectional sync from one replica to another would produce a false positive conflict at sync time t=6. The conflict arises from having two different versions at B and C at t=5 since the last sync. However, B's version at t=5 is derived from C's because C's mod vector is less than or equal to B's sync vector. As a result, Tra would replace C's version with B's without reporting conflict. Tra uses the same logic from Figure 9[1] to decide if two files are created independently, derived from one another, or conflicts using vector time pairs.

Figure 2: Tra No False Positive Conflict and Conflict Resolution Propagation



Conflict Resolution

At t=6 in Figure 2, we introduced a conflict by modifying the file on A and on C and syncing A to C. When a sync is run and the Tra server daemon notes the conflict, it asks to be sent the client's copy of the file. It then stores this file with the suffix '.tra_conflict_<REPLICA_ID>_<TIMESTAMP>'. This is akin to Dropbox, where conflicted files are given a new filename. On Dropbox, users are forced to manually resolve conflicts by updating and renaming the file they want to use and deleting all other conflicted versions of the file. However, our implementation of Tra contains a conflict resolution script that the user can run to automate this process. They only need to update the specific file that they want to keep, and run the script using that copy of the file as an argument. The script automatically removes all other conflicting copies of the file, and saves the resolution state to the `traCore` so that if we sync this replica to other replicas, they also receive this conflict resolution. In Figure 2, we resolved the conflict in favor of A. When A deletes the file at t=8, the change propagates to C at t=9 without repeating the same conflict.

Creation & Deletion Metadata

When a file is synced over from a remote replica, Tra differentiates whether that file is newly created on the remote replica, or has been already deleted on the local replica using the file's creation vector and the file's directory's sync vector.

Upon syncing, the Tra fs module detects that the file has been deleted if its metadata exists without the corresponding file node in the filesystem. Tra creates a "deletion notice" by

adding (or updating the existing) entry with the local replica's ID and the current timestamp in the file's parent's sync vector. It percolates the notice to the file's parent, who takes the max vector of its children's sync vectors along with the deletion notice and its own sync vector. Similarly, the mod vector of the deleted file's parent directory is updated by taking the max of its child and its own mod vector because "[a]s far as synchronization is concerned, deletion is just another kind of change.[1]" We update the deleted file's parent directory's mod vector so that the same sync call does not re-sync using the parent's old mod vector.

The original paper has a more detailed description of the logic behind creation and deletion in the form of pseudocode in Figure 10.

Rsync-like File Streaming

To cut down on bandwidth and improve performance, we use an rsync-like algorithm to propagate file changes from one replica to another for files larger than 4096 bytes. When the sync algorithm returns a result asking the client to propagate changes to a file on the server, the client queries the server for a rolling checksum table of its file. It then uses this table and computes a rolling checksum over its local copy of the file. If a local block's checksum matches one already present on the remote host, it simply tells the server to copy that block to the correct offset within the file; otherwise, it sends over the first byte of the block to the server, and moves the block over by one byte to compute a new checksum and try again. Our implementation uses the Adler-32 and MD5 checksums, just as rsync does.

Consistency Testing

To ensure basic functionalities, we replicated the figures from the original Tra paper[1] and another one we found[2]. For example, in Figure 3[2] from section 3.4, we made sure resolution state was saved by checking that the sync at t=4 reported no conflict when the resolution favored Y. Otherwise, if Z was favored, we verified there was a conflict.

For deletion, we ran the same tests from Figure 4 (a)(b)[1] where DEL means file is deleted. In Figure 4 below, our Tra implementation returns "Do Nothing" at t=4 when B ~> A because B's version is older than A's deleted version. In Figure 5, a sync in the opposite direction A ~> B at t=4 deletes the file on B. In Figure 6, A deletes and B changes results in a conflict.

Figure 3

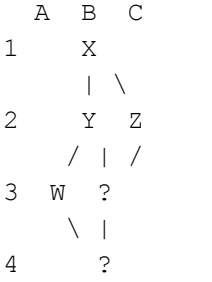


Figure 4

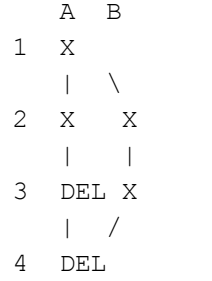


Figure 5

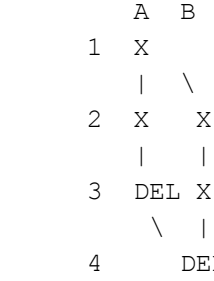
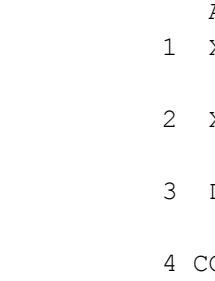


Figure 6

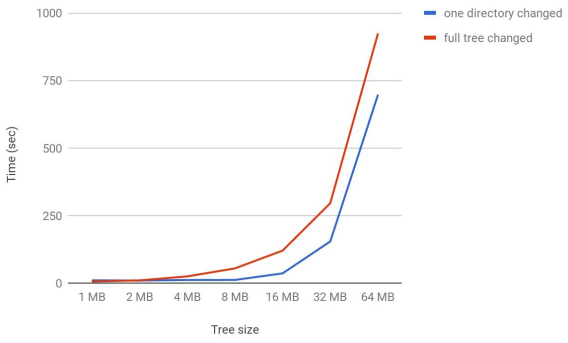


Performance Evaluation

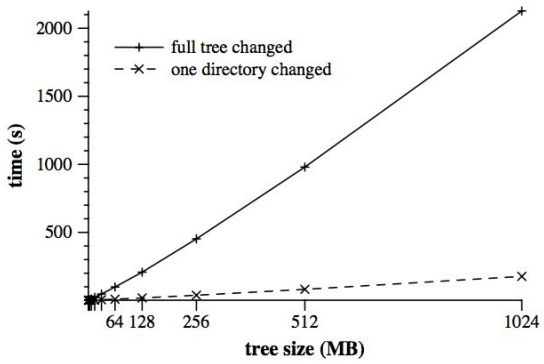
To evaluate Tra, we aimed to evaluate its “real-world” performance and verify that it satisfies the goal that runtime and network bandwidth is proportional to the size of the changed set. We created a N-megabyte file tree and ran Tra on two laptops, connected with an iPhone hotspot (802.11ac, >1Gbps). Each tree is constructed as a balanced binary tree of height $\log_2 N$ with N leaf directories, each containing 256 four-kilobyte files with random binary contents.

The following figures show the runtime for syncing a full tree update and a partial update (one directory changed). A partial update only updates files within one leaf node. We should see the runtime of syncing a partial update being 1/N of that of a full update. Our implementation of Tra is consistent with this result when the tree size is smaller than 32 MB. However, as tree sizes increase, the overall runtime does not scale up linearly. This is due to the overhead from serializing and deserializing for every sync command. For syncing up a partially updated 64 MB tree, deserializing metadata takes approximately 110 seconds and serializing metadata takes approximately 160 seconds. Since we populate all `traCores` and scan for any file updates before syncing, the time for serializing and deserializing is linear with the size of the tree, instead of the size of the change.

Figure 7: Runtime for Syncing N-Megabyte File Tree



Our Tra Implementation Results



The Paper’s Tra Implementation Results

There is a tradeoff between runtime and overhead for setting up replicated file systems. We implemented Tra in a way that minimizes the setup overhead for end users - i.e. they do not need to save to a particular directory, a la Dropbox. To achieve this, we design the daemon to store metadata on disk, instead of in memory.

We also attempted to partially replicate Figure 13 from the original Tra paper - comparing performance between our Tra implementation and rsync. The first test copied the Linux 2.6.5 kernel’s source tree from one replica to another. The second test synced the directories again. The third test changed a single file in a single directory and synced a third time. Again, we ran Tra on two laptops connected via an 802.11ac iPhone hotspot.

Figure 8: Raw performance comparison between Tra and Rsync

Time (s)			
>1000 Mb/s	copy	nop	change1
Tra	399.38	4498.24	2271.50
Rsync	16.3	1.98	2.66

Our Tra Implementation Results

	Time (s)		
	copy	nop	change1
100 Mb/s			
Tra	88.20	2.59	2.32
Rsync	34.73	2.45	2.34
Unison	67.86	2.05	2.67
1000 Mb/s			
Tra	61.14	1.69	4.67
Rsync	28.65	1.81	1.97
Unison	41.47	1.82	1.52

The Paper's Tra Implementation Results

The large performance deviations in Tra vs. Rsync are primarily due to the fact that whenever our implementation receives a sync request, it needs to unpickle the `traCore` from the directory corresponding to that request, check all subdirectories and files for modifications, and flush back to disk once the request has been serviced. This is why no-op takes a significantly longer time in our Tra implementation than rsync.

Future Work & Conclusion

There are also several ways we could improve upon this project. Firstly, our implementation uses system time instead of logical clocks because system time is easier to implement and debug. Logical clocks would have been a better solution because it would not require replicas to have the same clock. Secondly, we store metadata by directly serializing Python objects. In the future, we would like optimize both runtime and storage cost by storing only vector-time pairs. In particular, we used a modification vector instead of a scalar because it was simpler to implement. The storage cost for our implementation is therefore O(R*(D+F)) instead of O(R*D + F) from the paper, which implements this optimization. Lastly, we can optimize our runtime by using more efficient metadata storage instead of using Python pickles.

In this project, we successfully implemented a file synchronizer with no lost updates using vector time pairs. In particular, we achieved all the goals outlined above. We believe this is a good prototype of a user-friendly command-line tool file synchronizer across multiple replicas. The clear conflict semantics also allow users to select and resolve conflicts easily.

References

[1] http://publications.csail.mit.edu/tmp/MIT-CSAIL-TR-2005-014.pdf
[2] https://pdos.csail.mit.edu/archive/6.824-2004/papers/tra.pdf