TnT : A file system synchronizer [†]

Justin Holmgren holmgren@mit.edu

Zach Kabelac zek@mit.edu

Pritish Kamath pritish@mit.edu

Deepak Vasisht deepakv@mit.edu

May 12, 2014

Abstract

We build a peer-to-peer file-system synchronizer, based on Tra, which is performant and resilient to system crashes. We implement our file-system synchronizer in Go and ensure that all the guarantees for a file synchronizer mentioned in Tra are satisfied. We handle the system going offline by storing the metadata on the disk and using a two-phase synchronization method to ensure that crashes do not effect the system adversely even if they happen during sync. We robustly test TnT's correctness across 5 machines making 1 million directory changes and 1000 synchronizations across machines. Our results show that TnT successfully synchronizes across 5 machines.

1 Introduction

We implement a Tra based file system synchronization system. Similar to Tra, TnT stores time vector pairs for each file and directory. With these, TnT ensures all the desirable properties mentioned in the introduction of Tra, i.e. no restriction on synchronization patterns, no false conflicts, no metadata for deleted files, network usage proportional to changed files and partial synchronizations within the file tree.

2 Goals

We plan to implement a file synchronization system based on Tra. We plan to ensure that we achieve all the five objectives mentioned in the introduction of Tra Technical Report [1]. We will implement multiple versions starting from a basic version that just ensures correctness (including Tra's "no false conflicts" guarantee) and optimize further for reduced network communication, and reduced metadata overhead. We plan to test it on up to 10 machines (or virtual machines, maybe on EC2) running concurrent updates to file systems.

3 System Design

Tra introduced the novel idea of *vector time pairs*, which allows Tra to provide very strong guarantees such as *no false positives* and *no meta-data for deleted files* among others. Our main

[†] **TnT** stands for "**T**nT is **n**ot **T**ra"

system design is completely modelled after Tra. In order to improve upon Tra's robustness and performance, we create additional mechanisms as well. We improve upon Tra by designing a File Watcher that monitors the user's actions in the background. In addition, TnT is able to recover from machine crashes without losing updates becuase of it's Failure Recovery mechanism. We discuss these core aspects of our system below.

3.1 File Watcher

TnT structures synchronization meta-data in the form of a *tree*, which stores modification histories and synchronization histories for every file/directory in the file system. We use the `inotify` linux package, that allows the system to recognize when a user makes changes to the file system (such as creates/edits/deletes of files/directories) in an interrupt based manner. This allows us to keep the meta-data up-to-date with the latest state of the file system.

One design challenge associated with this method is that during synchronization, the File Watcher must be able to distinguish between whether the directory is modified by the user or by a synchronization. Since File Watcher is interrupt based, it must act based on the information in the interrupt. The `inotify` linux package returns the name of the file and an event mask describing the type of event that occured. The name and event do not provide enough information. In order to distinguish between a sync and user action, TnT first copies all of the changes to a *tmp* folder associated with its directory. Then, the process copies all of its changes from *tmp* into the actual location. This allows the File Watcher to distinguish between a change made by the user as opposed to the synchronization protocol. The *tmp* folder also plays a crucial role in implementing the 2-phase commit mechanism, described in Subsection 3.3

3.2 Synchronization

Along the lines of Tra, we consider only one-directional synchronization. Any two machines can synchronize their file systems independent of the rest of the machines. The vector time-pairs that we store in the meta-data allow us to check if there are any changes to be made in any particular file or inside the sub-tree of any directory. Thus, our synchronization protocol checks on the root if any change has happened in it's sub-tree. If yes, then we run the synchronization protocol recursively on all files/directories in the root. This ensures that the number of round trips of communication made is proportional to the size of the sub-tree that the two systems differ in. In particular, if only one file differs between the two synchronizing peers, then the number of round trips of communication required is equal to the depth of the file being sychronized.

3.3 Failure Recovery

We use a two-phase synchronization method to ensure that crashes do not affect the consistency of the system, even if they happen during sync.

3.4 Meta-data costs

4 Implementation Details

We will use Go for our implementation because of its good support for OS-level operations like watching the file system, high-level programming features, as well as our experience using Go in 6.824.

We will iteratively improve our file synchronization system, starting with a minimum viable implementation. In particular, for our first iteration we will only try to achieve correctness using essentially the same techniques (the Vector Time pair algorithm) as in the Tra, but we will not implement any nonessential optimizations.

If we have more time, we will add the optimizations described in [1].

5 Project Evaluation

We will test the system developed on up to 5 concurrent machines for:

5.1 Correctness

The most important aspect of a file synchronizer is that the directories be consistent across machines. To test the correctness of TnT, we test the case in which each machine independantly updates its respective file directory then randomly synchronizes with another machine. For this test, the machines are individual processes that watch over seperate file directories on one computer. The test will manipulate the files and TnT will synchronize it with other machines. The only way in which two machines can communicate is through the synchronization process.

The test updates a machine's files in a variety of ways on any branch of its directory. It creates and deletes files and directories, and also modifies the contents of files. The test does not check for renames or moves because TnT treats each of those actions as a delete then create. Once the updates and synchronizations are done, the test writes the directories, files and their contents to a string and hashes that string to a number. If the hashed numbers are the same, then the directories are all up to date.

The test has the machines update their file directories 200 times, then it synchronizes two randomly selected machines. The process of update then synchronize occurs 100 times totaling 100,000 directory updates per test. We ran this test 10 times and our results show that the hashes are the same for each of the 10 experiments.

5.2 Metadata overhead

For efficient operation, the metadata used at any point of time should be proportional to the size of the file system tree. When files are deleted by all the synchronizing peers, they should not contribute to metadata. To test whether the metadata overhead does in fact scale with the size of the file system, we measure the size of metadata for a machine over time against the number of files created. The measurement is taken during a normal experiment of with 5 machines where files and directories are randomly created and deleted.

Fig. 1 shows TnT's metadata versus the number of files created. The relationship between the metadata size and the number of files created is sub-linear as shown by shape of the

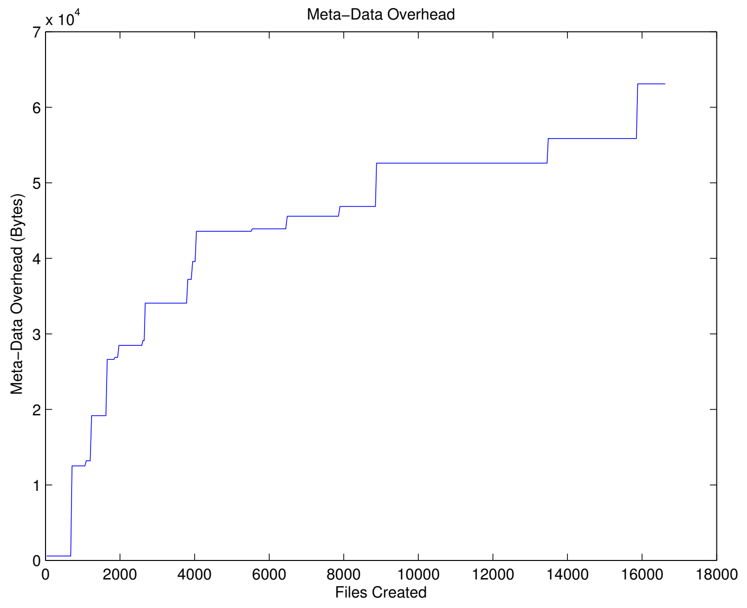

Figure 1: **MetaData OVerhead**. This figure shows the size of TnT's metadata as the number of files created in the distributed system is increased. The curve is sub-linear because metadata is proportional to the number of files in the system and not the number of files created. When files are deleted, they are removed from the metadata.

curve. This indicates that as the number of files and directories created increases, the size of the metadata does not increase at the same rate. Initially, many files/directories are created and few are deleted which explains the sharp curve. Once 4000 files have been created, the curve flattens out. The machine deletes more files and directories which substantially reduces the metadata. If the metadata stored information for each file created, it would not decrease in rate like that seen in Fig 1.

6 Acknowledgements

We would like to thank Russ Cox for helping us out with a technical doubt we faced in Tra.

References

[1] Russ Cox, William Josephson. Tra Technical report