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 11, 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. Slightly more stuff needed.

1 Introduction

We implement a Tra based file system synchronization system. Our implementation, by design, 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. The key

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 system design is completely modelled after Tra.

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

3.1 File Watcher

TnT stores 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.

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.

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 10 concurrent machines for:

- **Correctness**: We will test the case where machines modify disjoint sets of files, while reading all the files, and we will repeat this while changing the sets of files written by each machine, so that each file is at some point written by every machine. There should be no conflicts in this scenario, so we will check this, as well as checking that each machine has the same copy of each file. We will also check that whenever two machines concurrently update the same file in different ways, a conflict is reported.
- **Metadata overhead**: the metadata used at any point of time should be proportional to the size of the file system tree. if files are deleted by all the synchronizing peers, they should not contribute to metadata.

- ◇ **Communications overhead**: the communication used should be proportional to the number of files changed, and the sizes of the files. In particular, it should not be proportional to the size of the file system tree.

References

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