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

1 Introduction

We build a peer-to-peer file-system synchronizer, based on Tra, which is performant and resilient to system crashes. Similar to Tra, TnT stores vector-time 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. 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 1000 synchronizations across machines. Our results show that TnT successfully synchronizes across 5 machines.

2 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. In order to ensure robustness and resilience to crashes, we create additional mechanisms as well. On a high level, TnT's design can be understood in terms of two main primitives: first, we implement a File Watcher based on inotify to keep checking for file system updates when Tra is online, secondly, in cases when Tra comes back online after a crash, it recovers its metadata from the disk and checks if any file system modifications have been made when TnT was offline. We discuss these core aspects of our system below.

2.1 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

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

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 **??**

2.2 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.

2.3 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. The synchronization scheme goes through the following steps:

1. Update your metadata and record the changes to be made due to the current sync. Move files from the remote machine to the temporary directory and mark files to be deleted.
2. Dump the metadata to the disk.
3. Apply the recorded changes in the log and move files from the temporary directory to the working directory. Delete the files to be deleted.
4. Dump the metadata to disk.

This synchronization mechanism, on a crash, recovers the metadata from the disk and uses the metadata to update filesystem state if there are any changes that have not been applied. This ensures that crashes after/during any of these steps can be handled gracefully.
While recovering from a crash, TnT reads the filesystem state and figures out if any files have been modified since the last crash and updates their version vectors if that is the case. This

feature is enabled by explicitly tracking the last modified time of each file and directory in the metadata.

2.4 Meta-data costs

We managed to remove the metadata costs for deleted files. We had to seek help from Russ Cox to understand why his scheme works for avoiding meta data costs for deleted files and directories. Briefly, the synchronization vectors for a deleted file is the same as the nearest live ancestor and the modification vector need not be stored if we have creation time and the creator for the file stored in the meta-data. This helps us reduce the cost of storing meta-data as shown in the Results section. However, we do not implement the incremental version vectors and synchronization vectors that Tra does, so our meta-data scales worse than Tra.

3 Project Evaluation

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

3.1 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. We also tested TnT across 3 physical machines for consistency.

3.2 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.

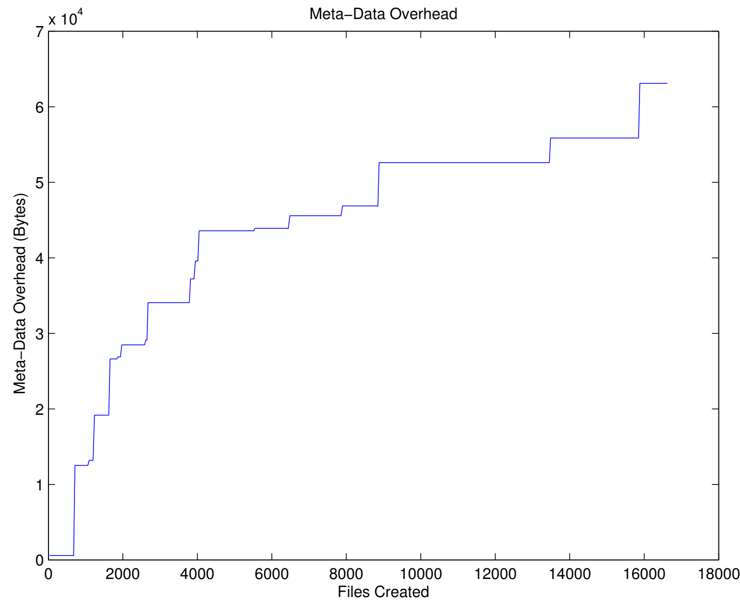

Figure 1: **MetaData OVerhead**. This figure shows the size of TnT's metadata as the number of files created in the distributed system is increased. The curve is sub-linear because metadata is proportional to the number of files in the system and not the number of files created. When files are deleted,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

Fig. **??** 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 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. **??**.

4 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