McSync
A File Synchronizer

Matthew Cook

January 13, 2014

Contents

Preface **7**

1 Introduction **9**

2 A Museum Analogy **11**
2.1 Synchronization 11
2.2 Inventories 12
2.3 Stages of a Sync: Continuity, Preferences, Upgrades 13
2.4 Sync-based upgrades vs. user-made upgrades 13
2.5 Records 14
2.6 Blathering on about records 15
2.7 The "dot & link" view of histories 16
2.8 When can an upgrade propagate automatically? 16
2.9 What information about the mesh do we actually need to store? 17
2.10 Complicated examples 18
2.11 Terminology 19

3 What does it mean to keep files synchronized? **21**
Intuitive Notions 21
3.1 Machines 21
3.2 Storage Device 22
3.3 Files & Aspects 22
3.4 Merge Tools 25
3.5 Tracked Files 27
3.6 Preferability 29
3.7 Scans 29
Comparing Versions 30
3.8 Full Histories 30
3.9 Comparing Full Histories 32
3.10 proof of sufficiency of "is same" approach 33
3.11 Optimized Histories 33
3.12 Comparing Optimized Histories 33
Advanced Notions 33
3.13 Gossip 33

3.14 Guidance - Identification . . . 35
3.15 Guidance - Preferred Values . . . 37
3.16 Noticing . . . 38
3.17 Instructions . . . 39
3.18 Syncing . . . 40
3.18.1 Updating a File . . . 42

4 McSync's Syncing Algorithm 43
4.1 Sequence of operations . . . 43
4.1.1 McSync operation . . . 44
4.1.2 The anatomy of a sync . . . 45
4.1.3 Older Versions . . . 46
4.1.4 Two Rounds . . . 47
4.2 Data Structures . . . 47
4.3 Formalized Algorithms . . . 48
4.4 Proofs of Correctness . . . 52
4.4.1 Lossless Theorem . . . 57
4.4.2 Optimality Theorem . . . 58
4.4.3 Consistency Theorem . . . 58
4.4.4 Local Improvement Theorem . . . 58
4.4.5 Global Improvement Theorem . . . 58
4.4.6 Naive Merge Theorem . . . 59
4.4.7 Ghost Deletion Theorem . . . 64
4.4.8 Free Preferability Theorem . . . 64
4.4.9 Arbitrary Distinguishability Theorem . . . 65
4.4.10 inode and ctime theorem . . . 65
4.5 Security . . . 65
4.6 Efficiency . . . 66
4.7 Discussion . . . 66
4.7.1 General comments regarding synchronization in general . . . 66
4.7.2 Quandaries . . . 67
4.7.3 multi-paths . . . 71
4.7.4 Beyond ASCII . . . 71
4.7.5 Extended Attributes . . . 73
4.8 Comparison with Other synchronizers . . . 74

5 How Syncing is Conceptualized for the User 77
5.1 Virtual Tree . . . 78
5.1.1 How the virtual tree lets you specify continuation and preference guidance . . . 79
5.2 Graft . . . 79
5.3 Backup Devices . . . 80
5.4 Propagation of Source and Configuration Files . . . 81

6 Mobility Issues **83**
6.1 separation of controller from large record keeping 83
6.1.1 Types of Processes 83
6.2 Hopping from machine to machine 84
6.3 Locations 84
6.4 McSync's Internal Network 86
6.5 Network and Remote Machines 88
6.6 Processes and Communication 89

7 Source Code Organization **93**
7.1 Preferences 93
7.2 Networking 94
7.3 Device Access 94
7.4 Data Transfer 94
7.5 User Actions Layer 94
7.6 Algorithm Layer 95

Index **97**

Bibliography **97**

Preface

McSync is a utility for keeping files on different machines synchronized ("synced") with each other.

How hard can file synchronization be? You have files on several devices, and you want to keep them in sync. Intuitively, you know exactly what you want. You want a system that propagates changes between devices, so that whenever you make a change on any device, this change will be propagated so that it shows up on all your devices. In cases of uncertainty, such as when conflicting changes are made on different devices before syncing them, you want the system to alert you to the situation, and allow you to resolve it easily.

Intuitively this seems to make sense. But creating an algorithm that matches this intuition in an ideal way turns out to be a much trickier undertaking than one would ever guess beforehand, as the length of this document attests to.

After reading about and/or trying every syncing utility I could find, it seemed clear that all the available options suffered from two flaws: failing to provide desirable usability features (such as those listed on page 9), and failing to have fully thought through the logic of syncing.

Almost all other tools use some overly simplified notion of syncing. This means that they report conflicts when there is no real conflict, or even worse, they fail to report conflicts that should be reported, leading to automated data loss due to incorrect automatic propagations.

The only solution to these problems appeared to be to write my own syncing solution, and thus McSync was born. (The first two letters of McSync come from my initials, M. C.)

This document explains what synchronization is, from McSync's point of view, and how it does it.

Chapter 1

Introduction

McSync is a file synchronizer that is:

- peer-to-peer (no centralized network or server needs to be accessed)
- multi-way (can sync any number of machines at a time)
- structure-flexible (synced files do not need to reside in a special folder (unlike DropBox or Google Drive), so your files can remain organized)
- OS-compatible (you can freely manipulate files with ordinary OS-provided commands like mv, cp, scp, or any other tools) (this is unlike e.g. git, where you need to remember to also tell git whenever you add or move a file)
- user-level (no special user priveleges needed; doesn't need to be built into the operating system)
- clock-safe (different machines do not need to have synchronized clocks)
- simple (no databases or other specially-installed software needed; can be used from the command line in a terminal window)
- open-source (gnu gpl, so you and others are free to improve it)
- optimal (reports as few conflicts as theoretically possible, fewer than any other synchronizer)
- optimistic (a technical term meaning that it propagates changes asynchronously, and is thus prepared for conflicts or sync failures)
- non-immediate (only syncs when you want; allowing you to change files and even resolve known conflicts while off-line)
- machine-flexible (different machines can store different subsets of the data)
- sync-flexible (any subset of the data can be synced on any given occasion)
- extendable (new mirrors can be added at any time)
- migratable (can sync data previously synced in another way)
- network-adaptable (storage devices can be synced even when attached to different machines at different times)
- file-following (renaming is understood as a type of change to a file)
- directory-following (rearranging of files and directories is understood as such)
- scalable (works fine with millions of files)

McSync has three fundamental innovations:

1. Its "virtual tree" interface lets the user easily organize what should be synced with what on various machines.
2. It was designed from the ground up to have all the properties listed on the previous page, yielding an unprecedented amount of flexibility.
3. It is built on a foundation of concepts that allow us to define precisely what it means to synchronize files.

The following chapters explain all of this in great detail.

Chapter 2

A Museum Analogy

2.1 Synchronization

You own a chain of museums, and you want to keep certain items, exhibits, or entire wings synchronized.

So if, while at one of your museums, you decide Rodin's Thinker should be rotated so that it is seen from the side as you enter the room, and you make it be like this, then the next time you sync the museums, all the other museums having the same room will also rotate the statue.

To help with synchronizing, there is a holodeck, which shows you the various wings of the various museums superimposed on each other (in the way that they are supposed to be the same, so a wing in the superposition will for the most part look just like all the corresponding wings in the museums do). This holodeck is organized at a high level in your favorite intuitive way, unconstrained by practical factors that individual museums have to deal with. For example, the holodeck might have a "non-art" wing, housing the cafeteria on the left and administrative offices on the right, even though these are in different places in different museums, and in many museums the cafeteria is nowhere near the administrative offices, and one museum even has two cafeterias (both matching the standardized cafeteria layout).

Walking around the holodeck, you can keep the real museums synchronized. You can walk to the holodeck cafeteria, notice that some museum cafeterias have the donuts near the coffee while others have the donuts near the soup, and specify that the donuts should be near the coffee. You can also just let changes propagate automatically, as in the Rodin Thinker example: Items which previously matched, but have been adjusted somewhere, will be adjusted in the other places as well to match the adjusted one.

2.2 Inventories

McSync shows you the holodeck. In each wing you see the parts of the museums that you have declared to correspond to this holodeck wing. But for the holodeck to show you the current state of these things, it needs reports from the individual museums on their current state. The individual museums compile inventory lists and give them to the holodeck, which shows all the museums together, pointing out where things have changed or don't match, especially if more than one museum rearranged the same room and you need to decide which of the rearrangements is better, or pick and choose among their changes.

If you think about it, an inventory list is a snapshot of a museum which could be in some state of flux. Although an inventory seems like a straightforward process, it in fact requires some intelligent oversight. For one thing, it's not a perfectly synchronous snapshot, because a worker had to walk from room to room to do the inventory, so if a painting was being moved from one room to another during the inventory, the worker might conceivably see it in two places, or not see it at all. Another complicating issue is that if a painting gets its data (e.g. oil on canvas, 1866, etc.) corrected (maybe moving to a different room), then it might not be clear, from mindlessly looking at the inventory list (which is the only information you have), that it is even the same painting. There are any number of things that could happen: If one of the museums starts retouching a da Vinci, you might decide that the museums should exhibit the untouched painting alongside the retouched one – the inventories should treat them as distinct paintings from now on. All of these issues point to the need for sentient oversight. No automated system can always know the right thing to do in these situations. As the holodeck guides you to places where things have changed, you can make executive decisions like "Yes, I see that the Rembrandt is gone, but I'm sure that's a mistake, please don't delete it from the other museums, but rather put it back where it was," or "This new red coffee machine by the cash register is actually an upgrade of the old green one by the door, go ahead and propagate this to the other cafeterias," or "I think some things have changed in recent minutes in the Stockholm museum, please redo their inventory of this room."

In the coffee machine example, what difference does it make to say the new coffee machine is an upgrade of the old one, rather than just saying the room lost an object and gained an object? Well, suppose a different museum had the good idea to label all the drink machines in Braille. If one museum upgrades the color and location of the coffee machine, while another museum updates the accessibility, these changes can be combined without any conflict. (Whereas if one museum discards an object while another museum upgrades it, then there is certainly a conflict that you need to resolve.) Also it is much cheaper for the museums following suit to move the coffee machine locally rather than throw it out and wait for a copy of the the new one from the museum that moved it. (Unfortunately, this is what most other synchronization utilities would do.)

2.3 Stages of a Sync: Continuity, Preferences, Upgrades

The first thing to do with an inventory is to confirm what current items correspond to what previous items. The museums keep records of their items, so when a new inventory comes in, we have to see which inventoried items are ones we have in the records, which ones are new, and which items from the records seem to have gone missing. As discussed above, this needs sentient oversight. Once this is sorted out (and only then), the museum can extend its records with the results of this inventory. This step takes inventory lists and old museum records, and generates new museum records.

The second thing to do is to specify, in holodeck places where the museums differ, which configurations are preferable over which others. This often requires sentient oversight. This step takes museum records and extends them with this preferability information (only needed for non-automatic preferences, it turns out). At this point some museums know that they should upgrade to another museum's version.

The third thing to do is to give the museums a chance to upgrade. Send them the new version and its history. If they succeed in upgrading to the new version (with its history), then this generates a bit more history which gets sent back around.

Then the sync is complete.

2.4 Sync-based upgrades vs. user-made upgrades

A user change to an object is the same as syncing with a transient replica that comes from the user's head. Since this replica has never been seen before and will never be seen again, we do not bother storing it in the history (although we could, for completeness), and the replica's history is of course only updated at the next scan, not at the exact time of the change. The replica appears, is marked as preferable to the existing copy, both are informed of the preference, and the existing copy takes the transient replica's value. Note that this preference was resolving a conflict, since the two histories were incomparable. So all changes in the system can be viewed as syncs, consisting of a preference plus propagating an existing value.

Or perhaps the transient replica inherits its value from the existing copy, gets changed within the user's head, and then a sync automatically propagates the new value back to the existing copy. But this approach leaves the upgrade as a different phenomenon from a sync.

What we haven't cleared up is what a preference is. But that seems plain: It must be an arrow pointing from the non-preferred version to the preferred version. McSync replaces one version with another whenever there is a preference chain leading from one to the other but not vice-versa. There is no difference between a sync-based upgrade and a user-made upgrade.

Note that if there is a value propagation, then either a preference was marked, or the value had previously propagated between those items. In other words, being at the same place in the holodeck will not make items sync for the first time – a preference also needs to be expressed, naturally, to know which one should change to match the other.

2.5 Records

An inventory list is a list of items (rooms, paintings, etc.). Each item has many property-value pairs: material: oil on canvas, size: 36"x27", year: 1876, title: "L'Absinthe", room: Impressionist Hall 2, etc. `http://www.musee-orsay.fr/en/collections/index-of-works/notice.html?no_cache=1&nnumid=1147`

A record is kept for each property of the item. The record shows the full space-time history of the property. This includes not only the time course (when it appeared in the museum and every upgrade it got since then) of that property of that replica, but also the histories of that property of every *other* replica that has been synced with this replica!

Why does an object's record include notes on objects it was synced with? Because that is how we can know whether these objects are in principle the same or not. If objects have been synced in the past, and are being synced again now, then we can check to see whether one museum has upgraded the property since a previous sync. (If so, the upgrade should be propagated. This is the basic job of a synchronizer.) Without knowing about the past sync, we can't know whether the other museum's version actually corresponds to this museum's pre-upgrade version (a clear prerequisite for propagating the upgrade), or just happened to be similar (due to chance) but not because the objects had been viewed as holographically the same.

If two objects are being synced, and one of them says it is made of clay while the other used to say it was made of clay but now it says it is made of porcelain, then should we upgrade the "clay" description to "porcelain"? Maybe, or maybe not. Perhaps the "clay" one was originally marked as "brick", then marked for a while as "porcelain", but now it has been changed to "clay". If the objects had been synced while both were in the "porcelain" stage, then we would know the "porcelain" one should be upgraded to say "clay". On the other hand, if they were synced while both were "clay", then the "clay" one should upgrade to "porcelain". Looking at the times of the changes might help, but different museums are not guaranteed to have their clocks in sync. In fact, even a single museum is not guaranteed to have an always-advancing clock. If the objects were never synced, or synced so long ago that they have both changed since then, then we can't be sure what to do.

We will see later that if two items were never synced before, then it can be dangerous even to assume that two items being synced, both labeled "clay", are indeed mirrors of each other! If one of the "clay" labels is treated as an upgrade over the other, then everything is safe (although more conflicts might be reported in the future due to this "modification" of the "obsolete" label),

because then there is a unique source for the label, but if one label can have multiple sources, then, surprisingly, upgrades can be lost!

2.6 Blathering on about records

Histories of other objects are only stored (by this object) up to the last time this object heard about the other object (perhaps via other objects). This information spreads whenever a preference is indicated, since a preference indicates that the two objects are (virtually) the same object, and whenever an upgrade is made. Being located at the same place in the holodeck is merely a tool the sentient overseer can use to help with choosing what to sync. For all we know, all the preferences we haven't heard about (due to being preferences between objects that this object hasn't synced with in a long time) are intended to be about what is from now on a virtually distinct object. Only if we are asked to sync with them would we assume again that they are the same.

So we see that the museum record is the maximal amount of information that that museum could possibly know about that object. If you go back in time, use a sync to cross to another museum record, and then go forwards in time, this is not part of the museum record, because we have no indication that the items are still considered the same. The most recent time in museum A's record, regarding the status in museum B, is at the most recent sync that leads (possibly through other syncs, but only forwards in time) to A's current replica. Anything since then in museum B might well be intended to be a forked version of the item.

Failing to indicate a preference when there is a conflict is interpreted as "maybe these don't correspond anymore", as that will have the effect that neither version will get lost (upgraded), which is arguably the correct behavior if there is a conflict but no preference was indicated. (*) (conflict)

So we have a history-mesh in space-time, labeled at various points with changes that were made (which can all be viewed as syncing changes, as described above). This history mesh, going back in time, is what is recorded in the records.

McSync happens to allow syncs only between items located at the same place in the holodeck (because otherwise things get very confusing), but the theory doesn't need this restriction. What a sync is is a declaration that item properties of a specific set of objects (usually in different museums) are "the same thing" and should match (this is nearly always done for all properties of an item simultaneously, but again the theory doesn't require this). The da Vinci example shows why we cannot just assume that two items should be the same – a sync represents a confirmation that this assumption is correct for a specific set of replicas at a specific point in their timelines.

2.7 The "dot & link" view of histories

The mesh is a bunch of snapshot dots connected with equality links and preference (arrow) links. Equality links are for when scans see the same thing again, or when a file is copied (by syncing) from one machine to another. Preference links are for when scans see that something has changed, or when a preference is indicated via McSync. Indicated preferences are committed as soon as the user presses go. They cannot be undone, although later preferences can render them irrelevant. When a new scan is done, a bunch of dots appear, and the identification step tells us where there should be links, and the scanned values indicate whether the links should be equality or arrow links. New items derive (with an arrow) from the previous implicit value of "missing", and items that have gone missing get a new value of "missing".

One version is preferable to another if you can get from one to the other via arrows (going the right way) and equalities, but you can't get back from the other to the one.

So when you resolve a conflict by marking a preference, it can't just add an arrow (because the reverse arrow might already be there, in which case the new arrow isn't clarifying things). It adds a new dot (for the version in your head), and an arrow from every extant version (same or not) you are resolving to the new one. This guarantees that the new version is preferable to the extant versions.

This "dot & link" view is almost the same as the traditional view, whose time-like paths of individual versions have been reduced to dots (or sequences of dots connected by equalities), and whose upgrade times have been expanded into arrows. Version propagations have become inter-replica equalities, and marked preferences have become inter-replica arrows. What's nice here is that marked preferences no longer need special treatment.

2.8 When can an upgrade propagate automatically?

Now, all we need to figure out is, given two history meshes for two objects to be synced, when is one version preferable to another?

What makes new files propagate? The virtual tree matches them up with locations in other replica directories. Any time "all the contents of a directory" are synced (via some interface command), then all the implicit missing files are synced as well, and their implicit histories implicitly match. This information would need to be stored somewhere, either in the notes for the directory (but that might never have been scanned) or in a multi-file record recording "missing" for all other filenames. You might even just scan one device, find an interesting file, and ask to scan that virtual file on all devices. If it comes back as "missing", that should presumably be stored in the history.

The only time we can know that mirror B is preferable to mirror A is if A was upgraded (perhaps multiple times) to get B. This is exactly the case when a

synchronizer should definitely update A to be B. All user changes are upgrades by definition, and the job of a synchronizer is that if a person upgraded a file on one machine, then the upgrade should be propagated to other replicas. Any time the synchronizer does an automatic upgrade, that upgrade can be thought of as a compressed sequence of human upgrades, each element of the sequence being exactly the same upgrade between the same two versions as what the human did.

This can easily be found by comparing the trees.

Type 1 comparison: If version A originated at a point listed in B's history, but B has been updated since then, then B is preferable to A. Note that if A is equal to any point in B's history, then A shares its origin with this point, so the origin is in B's history.

Type 2 comparison: If A's history indicates that some other version it knows about is preferable to A's version (this is rare, but maybe A hasn't been able to upgrade itself yet), and B's history says that B derives (perhaps with upgrades) from that preferable version, then B is preferable to A.

If versions A and B originated at the same place (with neither updated since then), then a sync simply gets them to share their histories. Or does this happen after identification, regardless of syncing? (*) (no conflict)

When an upgrade can happen automatically, no preference needs to be declared before the sync. The upgrade occurs because a replica sees a value/history combination whose history shows that that value is clearly preferable. So declared preferences are only needed in order to override the default (in)action.

2.9 What information about the mesh do we actually need to store?

Every point in a mesh has its origin in the mesh. But all origins in a mesh might lead to things outside the mesh.

For the type 1 comparison, we need to be able to know whether A's version originated in B's mesh, and whether A and B have the same origin. So we need to store the origin of a version. If the origin of A was on C, then we need to know whether this is inside B's mesh or after it, so we need to store some time anywhere from the last origin on C to the point C leaves the mesh.

For the type 2 comparison, we need to be able to know whether B's (non-green) mesh touches the green (futuristic) part of A's mesh. In other words, we need to be able to know whether any origin of a green part of A's mesh is an origin in B's non-green mesh. So for every green part of A (even multiple parts on one replica), we need to know its origin? (Then do a type 1 test for each.) It seems so. Any green origin could be the only one in B's mesh. How do green parts get added to A's mesh? By preferences getting indicated, or by hearing gossip about how green parts have evolved. Well, we only need the earliest green origin for each replica.

So, store own origin, and for each replica, let's store last non-green origin (if

any), and first green origin (if any).

Should we store preferences between other replicas? Something is green relative to an ancestor of ours?

Does the lack of a preference declaration before an automatic upgrade interfere with the utility of knowing you're obsolete?

For a naive merge, what history do we give it?

When a preference is declared, what do we do with the histories?

Any time we resolve a conflict or indicate a preference other than the default one, we are creating a "new" version, in the sense that it is preferable to all the versions it is replacing. Thus, it cannot have the history of any versions it might match, because none of those versions had a good enough history to be the default preference. It is like a "touched" version of whatever it matches, but touched to be ahead even of the things it doesn't match. It can come from a fleeting device in the user's head, but this device happens to derive from all the replicas being compared, as if the user's head copies a replica from each device, and upgrades all those replicas to be a copy of its extra fleeting replica.

2.10 Complicated examples

How could it be useful for an obsolete version to know that it is obsolete? We create (t1) some data V1 on A, mirror it (t2) to B, change it (t3) to V2 on A, and mirror it (t4) to C, meanwhile also changing it (t5) to V3 on B. Now (t6) we semi-sync A and B, resolving the V2-V3 conflict by specifying that A's version (V2) is preferable to B's (V3), but failing to update B. But B records this info, that (A,t3) and (C,t4) are preferable to (B,t5). Now (t7) if we sync B and C, then C has (A,t3,=)&(B,t2,≠)&(C,t4,=) while B has (A,t3,!)&(B,t5,=)&(C,t4,!), yielding A:"choose C", B:"dunno", C:"choose C", so we resolve in favor of C as we should. Obviously if B did not store any memory of the preference for A at t6, then we would instead have a conflict at t7 (the same one we already resolved at t6).

When users resolve conflicts or otherwise choose values that are not strictly preferable from the history, the resolved value actually gets a new history, so it *is* strictly preferable.

How can we see that declared preferences will override comparison problems? (Conflicts, reverse auto-pref, messed up hists?)

Are we sure about this new time for the conflict resolution superceding the chosen alternative? I mean, it solves some problems, but maybe it is too strong-handed of a solution and it doesn't really have full justification. Can't it lead to a conflict somehow that wouldn't otherwise be a conflict? Sure. A copies to C. A and C change independently. A copies to B. B makes a change. Now if A and B sync, B's change quietly propagates to A. But if A and C sync, there is a conflict. Supposed it is resolved in favor of A. Now if we sync A and B, we have another conflict. So syncing A-B then A-C presents you with one conflict, but syncing A-C then A-B presents you with two, the second of which is the same comparison that was resolved quietly in the first case. Is the second conflict

necessary? ??? I write most of a book and give it to my friend. We each finish it. I send my version to my editor, who makes some changes. I then see them (friend and editor) both at a cafe. If I talk to the editor first, the edits will get incorporated into my version. Then when I talk to my friend there is a conflict: we debate our endings. If I talk to my friend first then we debate our endings, and we choose mine. Even after this, it seems, the editor's comments should simply be incorporated, rather than treated as a conflict. Hmmph. ... Let's add the problematic case into that example, and see if we can't push the conflict to the point where the problem occurs.

Learning about a new museum displaying items seems separate from standard identification guidance, and could be transmitted at any time. ???

How does second round help? ???

Give 6-machine example where naive merge leads to upgrade loss. Point out workaround based on file update times: If a merge would take you to a version that pre-dates any of your naive merge sources, then it is marked as a conflict. Well, it has pros and cons.

As a simple example of what is meant by "optimal", consider syncing machines A and B (which are not connected to each other) via a USB stick U. You edit the same file on both A and B without realizing it, and then you sync U with A, so U updates to A's new version. Later, you sync U with B, and there is a conflict between the two new versions, which you investigate and resolve by saying that actually B's version is better (so U updates to B's version). Later you sync U with A again. Most synchronization utilities would report a conflict at this point as well, since again we have the same two new versions to choose between. But McSync won't bother you with this conflict that you already resolved. There are many more complicated examples, but the main idea is that McSync can remember what you told it, and it will bother you with conflicts as rarely as possible.

2.11 Terminology

Having come to the end, we introduce some terminology. Better late than never.

We need a name for an item in the inventory list (which is a single point in space-time), separate from an item in the museum records (which is a time-like path in space-time), separate from the holographically merged item visible in the holodeck (which is some collection of time-like paths in space-time, which are simply shown in the same place in the holodeck).

Item in inventory list: Snapshot Item in museum records: Replica Item shown on holodeck: Valuable

History, Mesh: Pedigree Instructions: Value+Pedigree pair Version, Aspect, Value, ...

Some valuables we like to keep in a state of inconsistent superposition of replicas. Other valuables we like to keep synced. Syncing is an operation we can apply to some or all of the replicas in a valuable, if we want. Creating a

valuable out of some replicas does not in itself mean that those replicas should be synced. That's what being "sync-flexible" is all about!

The intuitive notion of syncing can be used to motivate the notion of a valuable, and the notion of replicas. Considering how file systems and other utilities work forces one to break things down to snapshots.

Chapter 3

What does it mean to keep files synchronized?

Intuitive Notions

3.1 Machines

It is useful to distinguish between machines (computers, perhaps in the form of tablets, phones, servers, or other devices) and storage devices (hard drives, flash memory, SD cards, cloud storage, and so on).

Machines only have three uses:

1. They have CPUs which can run McSync.

2. They can access data on storage devices: not just files to be synced, but also the McSync executable to be run.

3. They can communicate with other machines at other locations. This allows data to be transferred between storage devices.

Everything McSync knows is stored on storage devices. Nothing is stored on a machine – storage requires a storage device. Even the name of a machine is given either by data on a storage device (point 2 above) or by a network address (point 3 above). A machine without a storage device or a network has almost no distinguishing properties except for its cpu (point 1 above), which determines the executable that will need to be compiled and used.

Anything a machine knows is like short-term memory, only useful for current operations. Only a storage device has long-term memory.

3.2 Storage Device

A storage device is something that can store data, specifically, the directories and files of a file system. This could be a USB stick, or a drive with multiple partitions mounted, or a computer with one or more hard drives, or a machine with multiple physical storage locations mounted. What is important is that the files on a single storage device must always lie in a single directory tree. The file locations must be fixed relative to the McSync directory on that device. So removable drives should be their own device, as should directories that might get mounted on different machines.

Each storage device has a single McSync directory where McSync stores all its internal files (executables, logs, scans, etc.). The device can be fully accessed by a single executable on a single machine (although it may well be on different machines at different times, as with a USB stick, or a directory mounted from a server). We will freely speak of the device doing things, when in fact of course the machine that the device is currently attached to is part of what is actually getting the things done.

The internal files store both general McSync files, like executables and documentation, and device-specific information.

The following things are unique to a filesystem:

- its id (identifying the filesystem)
- info about user names and group names (e.g. user jdoe on device A might be equivalent to user john on device B)
- its time (an increasing integer, used in the histories)
- its histories (history5, history6, etc.)
- a (usually absent) lock file (only one instance of McSync should be manipulating the internal files at any time)

Each device has a device id that McSync tries to make unique on every device (of course not guaranteeable in principle).

3.3 Files & Aspects

By "file" we mean "file or directory". To refer to a file that is not a directory, we say "plain file" or "leaf".

A file is no more than a collection of aspects, like the name, owner, modification time, etc. The contents of a file are just another aspect. But to record the contents of such unlimited-size elements (for comparisons), we just store the md5. Resource forks are just another aspect of a file, like the regular contents. The parent directory is another aspect. There can be an arbitrary number of these, just like forks. The path is not stored, just the parent, which is stored as a McSync file tracking number (see below).

There can be an arbitrary number of aspects. Each extended attribute (as in getxattr) is an aspect. Unlisted aspects are aspects that have been "missing" since the dawn of time.

- "Missing" is a value that an aspect has if, according to the file system, that file does not have that aspect.
- "Notvisible" is used for an aspect whose value presumably exists, but cannot be accessed (typically due to permission problems – can happen for file contents, or for certain extended attributes, or for everything). "Notvisible" is not a real value that an aspect can actually have.

The aspects used in McSync are:

- filename
- parent directory
- contents
- modification time
- security (owner/group/permissions)
- any extended attributes

Names do not include the path. Paths are not explicitly stored, although they are implicit via the names along the chain of parent directories.

Note that only leaves can be multiply-hard-linked. (Only leaf inodes can be referenced from multiple filenames, unless root has been using link() to mess things up.) Exceptions:

- Mac's TimeMachine drives – we will ignore these.
- . and .. are hardlinks to directories.

So it is correct to think of the directory structure as a tree. McSync treats hardlinks as separate files which happen to share certain aspects, so syncing with one file can also update part of the other's history (e.g. for content updates but not for filename updates).

Pretty much all aspects are given by the inode. The only thing not from there is the name and parent directory.

The contents of a directory, i.e. the list of names and inode pointers, are each treated as aspects of the corresponding file, rather than as aspects of the directory itself. Otherwise, there are pointless conflicts when two machines make changes in the same directory. So the directory itself doesn't have any "contents" aspect. It has most other aspects (permissions, ownership, modification dates, etc.).

Symbolic links are treated like any other file, except for the link contents. While directories have no syncable content, and files have the standard syncable

content, symbolic links have two syncable content aspects: full vs relative, and pointed-to-file. The main link content, the pointed-to-file, is maintained by McSync in a special way that is different from standard syncing, since in general one really doesn't want directories getting renamed just because you changed the contents of the link: Movement (due to syncing) of the pointed-to-file propagates into the link contents, but changes in the link contents don't propagate beyond the symbolic link itself. So if there is a symbolic link to /usr3/cook/ on one machine which on another machine is a link to /Users/cook/, then these contents won't be touched when the links are synced. But if the target directory is renamed to cook2 (and this change is tracked by McSync), then these links would be updated IF the virtual tree had the link content grafted onto cook. And if you changed one link to /usr3/cook/temp/, then the other link would change similarly when synced (no graft required, this is a sync between two pointed-to-file aspects). The link's contents (target path) are kept synced with the actual target filename and all directory filenames on the relevant path, for all such parts as exist in McSync's histories, IF the virtual tree has the link content grafted. On each device, the goal is to keep the link pointing to the given tracked file (if the link itself didn't change). If the link itself is moved (in a way McSync can detect), and the link is relative and would need to change, then its contents are updated accordingly. McSync would warn if a grafted link was edited, proposing re-grafting rather than syncing the true file location to the new symbolic link contents. Of course, the user can see if link modifications are scheduled due to directory rearrangements or renaming, and can indicate that the link contents should or should not be updated, for any link or subtree. Unupdated links will not remember that what they point to has moved. Symbolic links are never followed in the usual sense.

Sometimes certain filesystems can't represent certain aspects of certain files.
Non-macs: resource forks, inode Btime
Macs: symlink dates
(Note that these are properties of the filesystem, not really of whether it is a Mac or not.)

For these cases, we consider the unstorable aspects to have the value "unstorable". The aspect does not exist on that machine, and does not take part in syncs. It is as if that single aspect is pruned from that image in the virtual tree. So you can't sync resource forks across a non-mac bridge. Seems ok.

Can this be the same approach as is used for parts of the tree pruned on a certain machine?

Yes, the quandaries are solved by "unstorable" being different from "missing", since only the latter indicates a deletion. A file that has been moved within a remote graft, but is now outside the current graft, gets every aspect marked as "unstorable". The value "unstorable" does not have an associated adoption time, whereas "missing" does. "Missing" means it was deliberately removed, whereas "unstorable" is a disability of the file system or graft leading to a lack of knowledge about this aspect, much like "notvisible" (which results from uncooperativity of the filesystem).

Certain filesystems may have "coordination restrictions" on what can be stored. For example, you typically can't delete a file's name or permissions without deleting the file's contents: different aspects are forced to coordinate on the value "missing". Another "co-res" is that two files in the same directory cannot have the same name. Another that only applies on some systems is that two files cannot coexist if their names match except for case, or perhaps if they match for the first 8 characters. Co-resses interfere with storing arbitrary data. If a co-res exists on A but not on B, then it can be impossible to replicate B's state on A. In these cases, McSync will arbitrarily choose some of the data to be "unstorable", and might add temporary prune points to the graft for that device.

Deleted files are considered as "ghosts" for a while, files that are "present" but for which every aspect has the value "missing". This allows deletions to propagate properly.

We have three special values for aspects: "notvisible" data may well be there, but cannot be read (a permissions problem, or a mounting problem) "unstorable" data cannot be written (device doesn't store such data, or it would lie in an ungrafted area) "missing" data is deliberately absent, even a complete replica of the file will have nothing stored for this

Is there any difference between "notvisible" and "unstorable"? For the algorithm, no, because the point at which the algorithm is blocked from the data is irrelevant (we cannot read and/or write for whatever reason). For the user and what they will do about it, yes, they are quite different. The third, "missing", is more like a normal value to be synced and compared with other values. "Notvisible" and "unstorable" are things that are noted about a file aspect, while the aspect itself may still have a known value (or hash) in the history.

3.4 Merge Tools

Merge tools are data-structure-aware algorithms for deciding what guidance to give.

What does it mean to sync two objects? Say drive A with drive B? One person might erase one drive and copy all the bits from the other drive (and if both drives were updated, mark it as a conflict), while another person might claim there is internal structure to the bits, and something more intelligent can be done. So what does it mean to sync two things with internal structure? It depends on the structure of course. A semi-intelligent entity, that understands something about the structure of the contents, must break the structure down into substructures that can be handled by other entities.

McSync can use plug-ins for various types of files. For example to do line-based code merging, or freemind tree merging.

What such a plug-in does is to replace the idea of syncing A and B with the idea that A has parts and B has parts and these parts can be synced. Tracking changes in the parts and rearrangements of the structure is the job of the plug-

in. In general, "location in the structure" is an aspect of a part, thought of in the most local way possible. A file has "location:parent" and "location:name" as the two aspects of its location in the directory structure. A code file might have "location:previous line" as the lone aspect for each line in the sequence-of-lines structure. Of course, intuition-matching heuristics are needed to make a reasonable interpretation of observed changes, in terms of objects and their aspects, and it must be possible to provide guidance to correct the heuristics on any given occasion.

One important type of structure is a dictionary with key/value pairs, and different keys have different meanings (structures for the value), so it doesn't make sense to move a value to a different key. (For example, aspects of a file are like this. In normal operation, it makes no sense to say a value was moved from the "owner" key to the "permissions" key.) Items in such a dictionary are synced key by key, syncing values with other values in unsynced places having the same key.

Another important type of structure is a set structure. (The set of filenames hardlinked to an inode is an example of a set.) Here there are no keys to tell us whether a member has been edited or deleted and another one added. One simple solution is to consider the elements as keys, and the presence or absence (0/1, or more if a multiset) as the value. This would reduce the problem to merging changes in an integer, but this reduction of the problem is not always the right one. For example, if foo.txt gets renamed on machine A to foo1.txt and on machine B to foo2.txt, most users would expect that to show up as a conflict, because the single item in the set was edited on each machine, rather than saying there is no conflict because both new filenames should be hardlinked to the same file and the old hardlink removed. If a single file had names A1 A2 B1 B2 and on one machine they changed to A1 A2 Z1 Z2 and on another machine they changed to A1 A0 B1 B0, then is there a conflict? Almost certainly, because it is too confusing for any solution to be clearly correct. We want to interpret changes on a given machine as adding links (new is superset of old), deleting (new is empty), reducing links (new is nonempty subset of old), renaming (new and old are each size one), or other (new and old are each at least size 2 and there is at least one possible rename). "Other" conflicts with everything (except an identical "other" or "nothing" of course).

```
conflict?   add   delete   reduce   rename    other
   add       ok     X        ok       ok        X#
  delete     X      =        ok       X         X
  reduce     ok     ok       ok*      -         X#
  rename     ok     X        -        X         -
   other     X#     X        X#       -         X#
*ok if the combination does not result in total deletion
-can't happen
#ok if changes on one machine (this link gone, that link new, etc.) are a non-strict su
```

In fact, that description handles everything except add-add, reduce-reduce*, rename-add, ...

If a machine just adds, the new ones are new.

If a machine just reduces, the old ones were removed.

If a machine renames, the entry was edited.

Extend the definition of rename to also include cases where new and old restricted to a given directory are each size one, and do these renames before checking "just add"/"just reduce".

So if a file has more than one hardlink, and you move it to a different directory or different name in dir containing another link to the file (or do some unlinking and linking that could be mistaken for this), this is the only case where we are not sure whether it is a rename or add/delete. To make heuristic cover all cases, an add/del pair is a rename iff their deepest shared ancestor directory has no other instances.

A hardlinkable file provides a 0/1/+/- vector (indexed by machine) at its points in the virtual tree. A directory takes a vector from each child, and if (1,1,-,0) meets (0,0,+,1) then it's a rename and (1,1,1,1) propagates up.

This is mind-numbing – must stop and come back after data structures are clear for merger (histories,scans) $\longrightarrow$ (guidance).

options:

add-add (ln a b vs. ln a c)

reduce-reduce=delete (rm a b vs. rm c)

File contents are similar, but the keys are simply a linked list of lines, and the number of lines in a region can grow and shrink. For a set, we cannot do this clean divide-and-conquer.

The file system as a whole is a big tree structure with vertex names, and much of McSync is devoted to doing the right thing in this case.

3.5 Tracked Files

The word "file" is unfortunately the natural word for at least four different concepts.

1. Raw files. A device contains raw files. These are files in the standard UNIX sense. It helps to imagine that these only exist (with a particular name / contents / other aspects) when you look at them, since users and programs may be changing/moving them at any time. Only the operating system can monitor all changes to a file; user level programs can only take snapshots of the raw file's state at any time. And even the OS can only do it for local disks. To be compatible with existing operating systems, McSync works with snapshots.

2. Tracked files. On each device, McSync keeps track of files. A tracked file corresponds to a sequence of snapshots that have been determined to correspond to the same conceptual file. A tracked file is not a raw file, since its name/locaion/inode might change over time for example. Tracked

files are also called replicas. Replicas are what can be synced with each other.

3. Virtual files. A virtual file is a location in the virtual tree, corresponding to multiple tracked files on multiple devices, which are kept synced with each other. A virtual file feels to the user like a single file, stored in multiple places. No records are kept on virtual files. From the algorithm's point of view, there are only tracked files, which get synced with each other. Only the interface uses virtual files, to help the user organize their thoughts.

4. Ancestors. This is not a standard concept. These are previous or remote versions of aspects, as listed in McSync's history records. Actually, even the current local version is listed as an ancestor.

McSync uses its own tracking number to identify files. A tracking number represents one tracked file on one device. At any instant, it corresponds to at most one file on the device. A tracking number is the same as a tracked file, although one refers more to the id number and the other refers more to the intuitive feeling (having conceptual continuity) of the whole file.

A tracking number includes the device number. (This might be only implicitly represented in the code, since many tracking numbers typically share a large constant device number.) When the device number is explicitly included, we call it the global tracking number.

McSync uses its own file tracking numbers on each storage device to uniquely identify a file, regardless of changes in location, name, or inode, or anything/everything else. The inode is not an aspect of the file, because it has no inter-device meaning (and thus cannot be synced), but it is recorded nonetheless (just like ctime, btime, etc.) to help track down what happened. The history of the files on a device contains a history (defined below) for each aspect of each file tracking number.

At any given instant, a file system contains files in the way determined by the data and format of that file system. But in our minds, files are things that persist over time, in a way that the file system is unaware of. For example, many applications, when saving a file, actually create a new file and then remove the original (to reduce the danger of data loss), but as users we view the new file as the unique continuation of the old one. McSync's job is to track files as they change over time, and so it thinks of a file as something that persists over time. This is why it needs to use its own tracking numbers to represent conceptually persistent files – because the file system itself cannot maintain identifiers whose persistence corresponds to our intuitive notion of "the same file".

Although McSync will follow a tracked file as it evolves on a single file system, it does not view other files on the same or other file systems as also being the same. When we sync two files, whether on the same or on different file systems, they have distinct tracking numbers. Syncing is an operation in which file aspects get copied from one file to another. McSync records the fact that certain file aspects were copied from one to the other, but their file tracking numbers remain distinct. In other words, there is no notion of an inter-system

file. The virtual tree is the structure for helping to figure out what files on one system should be synced with what files on another system (and the histories of individual files also have related information), but there is no explicit record anywhere of all the files in the virtual tree – there are no virtual files. A file exists on a single filesystem, and has a unique state at any given time. Any two files can be synced at any time, and this sync will be recorded in their histories, but it has no effect on the fact that they are two separate files.

3.6 Preferability

An updated version of a (tracked) file is to be preferred over an old version. If you doubt this, then you don't want a synchronizer. This is the one and only idea underlying what it means to do synchronization.

What a synchronizer does is to replace non-preferred versions of files (or file aspects) with preferred versions.

When a user edits a file, we consider this action to demonstrate the user's preference for the new version of the file.

Preferences are discovered by observing changes on disk (via a scan), and they can also be indicated explicitly in the McSync user interface.

Preferability is transitive. If A is preferable to B, and B is preferable to C, then A is preferable to C. The same goes for "preferable to or equal to".

[[discuss preferability given full history. show that only our data structure is needed. but our data structure isn't introduced yet.]]

When comparing two histories, the files are equal if any of the "issame" parts of the histories match. File A is obsolete compared to file B if there is any machine where A shows "same" but B shows a later adoption time or an equal one but is not the same. If more than one of "same", "less", "greater" seems to hold, a conflict is marked. If there is a version that a machine is purely obsolete with respect to, then the machine will get instructions instead of gossip for this item.

When looking at a file aspect on two mirrors A and B, A is preferable to B if there is any mirror C (possibly A or B) which is listed in B's history as being identical, and which appears in A's history with a time equal to or later (newer) than how it appears in B's history. If the time is equal and A's history also lists it as identical, then this aspect is identical on A and B and needs no update. (If it is not identical for some reason, or if both are preferable, at least one strictly, then a warning/conflict is displayed.)

3.7 Scans

A file or directory can be scanned at any time. This simply means looking at its aspects, and generating a "snapshot/fingerprint" record, which is a timestamped snapshot of every aspect of the file except the large ones, which get a fingerprint. This snapshot/fingerprint is called a scan.

file.txt: modtime:04:22, name:"file.txt", owner:"www", md5:blah, inode:5757, etc.

Note that the scanned files do not have tracking numbers. They only get tracking numbers later, when they are matched up with tracked files (which can require assistance from the user).

Typically many files are scanned shortly before synchronizing. This mass-scan is also called a scan. Such a scan typically scans all the relevant files on the storage device, once, regardless of how often they appear in the virtual tree. Thus, additionally syncing another part of the virtual tree may lead to only a few or no new files needing to be scanned.

A scan belongs to an actual file that is on a particular file system. A virtual file cannot be scanned (although you could ask to scan all the actual files corresponding to it on all the machines you are connected to, an activity which you could reasonably refer to as "scanning that [virtual] file").

Of course, the user can ask to rescan (or not scan) any portion of the virtual tree at any time, on any or all reachable storage devices.

A scan takes place at a certain "scan time", which is given by a device-specific integer that increases with each mass-scan. All the syncing algorithm needs is for each aspect of each file to have such an integer, but we use a common integer for all scanned file aspects in a mass-scan, to make it easier for the interface to figure out when the scan was. We could use the absolute time, but that would force us to trust the system clock (which might be on different cpus for different scans of the same file system), or use a combination of absolute and increasing time. Presumably an incremented integer could allow much better compression than an accurate time, which would be different for every file in a mass-scan.

Note that one file on a device might be scanned at t=16 before another file gets scanned at t=15, for example if the device is following instructions, and using t=15 for the pre-update check that nothing has changed, and t=16 for the post-update check that the update worked.

The scan will of course use the mtime etc. flags to optimize away the reading in of the whole file and recomputation of the md5.

If a directory is accidentally set to be unreadable, there will be a problem with that part of the scan, rather than it looking like everything was erased. The unreachable file aspects will be noted as "notvisible".

The purpose of a scan is to be merged into a history. But to do this correctly requires guidance. We describe histories next, since they provide the background of what gossip and guidance are for.

Comparing Versions

3.8 Full Histories

Each storage device maintains a history for each file.

Histories are somewhat similar to the well-known concept of "version vectors", and even more similar to the more recent concept of "vector time pairs".

However, histories are designed to work in a decentralized context. There is more on this in the "relation to previous work" section below.

[see figure] [figure shows tracked file lines on multiple devices, connecting snapshots. Some new snapshots need to be connected by identification guidance. A history connects multiple lines, can be on same or different devices. The connections are horizontal (if the lines were vertical) n-way connections – these are the syncs. Snapshots are marked where a change occurred. The history of a tracked file goes up lines and across syncs, ending at each snapshot of change.]

Every aspect of every tracked file has a history. The history is a list of ancestors. Ancestors are aspects that are the same version or an earlier version of the current local one. The most recent ancestor is recorded for each other tracked file that the aspect has ever been synced with. Specifically, the (scan) time of adoption (by that tracked file) is recorded, and the "sameness": whether the version is the same vs. prior to the local version.

Histories could record more, much more, but surprisingly it turns out that this is all that needs to be known. See the Optimality Theorem.

Histories do not keep track of alternate versions (which might after all not be the same virtual file anymore in the user's mind). A file's history cannot know that the file is out of date or that there are any branched versions of it. The history is the list of ancestors of the file that is there, not some abstract history of related files it might or might not be synced with in the future.

Each aspect has a list of (storage device : file tracking number, latestchange, issame) records. Each storage device has its own file tracking number assignment system, so (storage device, file tracking number) makes a global file tracking number. Each device keeps track of "time", a device-specific integer that increases with each scan. latestchange is the birthday (first scan time on that device when the new value was noticed) of the latest value of this aspect that we know about on that device. issame is a flag that says whether the aspect has changed at all (even if changed back) since then. Any value aspect that a history knows about is always one that is strictly previous to (has been updated to) the current aspect.

Example: file tracking number 72: name:"file.txt",(m1:56,17,no),(m2:72,12,yes), owner:"cook",(m1:56,14,yes),(m2:72,12,yes) (say this is the scan on machine m2)

Then we chown the file on m2. On the next sync on m2, which has an m2-time of 15, the entry is modified: (highlighted here with <>) file tracking number 72: $name$: "$file.txt''$, $(m1 : 56, 17, no)$, $(m2 : 72, 12, yes)$, $owner$: "www'', $(m1 : 56, 14, <\ no\ >)$, $(m2 : 72, <\ 15\ >, <\ yes\ >)$ This line can be interpreted as saying that the name has changed since m1's latest name-setting time (that we know about), which is $t_{m1} = 17$. And m2 picked up the current name on scan $t_{m2} = 12$. The latest owner-setting time for m1 was $t_{m1} = 14$, but we have a newer owner than m1's version (if m1 downloaded the scan and synced the owner to our new value, we don't know about that yet). We, m2, picked up the current owner at $t_{m2} = 15$. All times are device-specific.

If all the info for a ghost file indicates that it has been deleted (all aspects "missing") everywhere, then it can be removed from the history for good. (Proved in proof section below.)

The history list may be of a different length for different files or different aspects, because the user on some occasions might just sync the name but not the permissions, or whatever.

A new file would have for every aspect a list of length one: (the present machine, the current time, yes). This is set($\emptyset$,t).

In general, each aspect has a list entry for every device in the past light cone of the file, giving the time of the last change (on that device) before the cone boundary and whether the file at that point was the same as it is now. The light cone is defined in the web of gossip links and tracked-file edges (both directed by time). Note that if the file at that point differed in any aspects from the one now, then the one now is a strictly newer (preferable) version. This is discussed much more under "Proofs of Correctness".

The light cone: Only gossip connects spatially different points (different tracked files). If a file is noticed to be changed but is not then synced, no synchronization occurred, and the histories do not assume that the two files should be synced, i.e. they do not assume that they still correspond to the same virtual file – only a sync confirms this. A sync causes a history to include ancestors of the previous version as well as of the new version, but all communication between tracked files regarding ancestors is gossip.

Up until the sync, it could be the case that the user has decided to fork the two actual files, and they should never be synced again. In fact, this is the default assumption. (Even a conflict-free update needs to be approved in some sense by the user.)

The purpose of a history is to enable the algorithm: If two files have a common ancestor, and one is unchanged since then, it updates to the more modern one IF/WHEN the two files are synced again.

3.9 Comparing Full Histories

There are a few relations/operations that are defined on histories.

First note that when comparing (tracked file aspect A, t=N1, issame=B1) with (A, t=N2, issame=B2), we are comparing the numbers N1 and N2, with the caveat that if issame=NO, then that adds an arbitrary amount (0.01 to infinity) to the corresponding number. So we have:

Comparison:

	ab	3=	3≠	4=	4≠	5=	5≠
ab	?	?	?	?	?	?	?
3=	?	=	↑	↑	↑	↑	↑
3≠	?	←	?	?	?	?	?
4=	?	←	?	=	↑	↑	↑
4≠	?	←	?	←	?	?	?
5=	?	←	?	←	?	=	↑
5≠	?	←	?	←	?	←	?

ab = absent

We can see an absent record is like $(-\infty, \neq)$. If the lower number has an =, then it is less. If equal numbers each have an =, then they are the same. Otherwise they are incomparable.

$h1 \sim h2$ At most one of $=$, $<$, or $>$ holds among all $(h1.t, h2.t)$ for tracked files present in both histories.

If $h1 \sim h2$ does not hold, that is a conflict. This could happen if a conflict is resolved different ways in different places.

$h1 > h2$ $h1 \sim h2$ and for some tracked file present in both histories, $h1.t > h2.t$.

$h1 = h2$ $h1 \sim h2$ and for some tracked file present in both histories, $h1.t = h2.t$.

$h1 \geq h2$ $h1 > h2$ or $h1 = h2$

$h1 + h2$ Addition yields a merged history. It is like a termwise max, with $\neq$ worth 0.5. But terms can interact slightly:

If $h1 > h2$, then $h1 + h2 = h1 + post(h2)$. This part of it is not termwise. The rhs, $h1 + post(h2)$, can be done termwise.

More generally, if $h1 \sim h2$ does not hold, only those histories go un"post"ed which are in no term less than any other history.

However, if there are multiple such equivalence classes (as in a naive merge), only one may go unposted.

Addition yields the history to use when ancestors had the summand histories.

Addition:

+	ab	3=	3≠	4=	4≠	5=	5≠
ab	ab	3=	3≠	4=	4≠	5=	5≠
3=	3=	3=	3≠	4=	4≠	5=	5≠
3≠	3≠	3≠	3≠	4=	4≠	5=	5≠
4=	4=	4=	4=	4=	4≠	5=	5≠
4≠	4≠	4≠	4≠	4≠	4≠	5=	5≠
5=	5=	5=	5=	5=	5=	5=	5≠
5≠	5≠	5≠	5≠	5≠	5≠	5≠	5≠

ab = absent

post(h1) This unary operation simply changes every $=$ to a $\neq$.

set(h1,t) This binary operation sets h1.t to the current scan time, with $=$.

3.10 proof of sufficiency of "is same" approach

equivalent to full history approach

3.11 Optimized Histories

3.12 Comparing Optimized Histories

Advanced Notions

3.13 Gossip

There are only two kinds of information that gets sent between devices (specifically, between tracked files). One is that during a sync, the actual values of aspects are transferred. But the other is that ancestor information (history information) gets spread around. Even when actual aspect values are transferred, they are accompanied by the history of the new value. Sending of histories

between tracked files is called gossip. Gossip provides the spatial component of the light cone.

The idea of gossip is simply that an ancestor of an ancestor is also an ancestor. So any information about their ancestors that you hear about from your ancestors can be incorporated into your own ancestor information.

If $t.h >= s.h$, then $t.h$ can update its history to $t.h + s.h$. This is the gossip operation. It transports information between tracked files.

Gossip is when two file aspects can tell from their histories that they are related.

mirror A: file1: contents: p (A:file1, 3, y)(C:file1, 5, y)

Say we have that history on A, and one of the following histories on B:

mirror B: file1: contents: q (B:file1, 9, y)(C:file1, 5, n)

mirror B: file1: contents: q (B:file1, 9, y)(C:file1, 6, n)

mirror B: file1: contents: q (B:file1, 9, y)(C:file1, 6, y)

In these cases, B is preferable to A, and B can add (A:file1, 3, n) to its history, but A cannot add anything.

mirror B: file1: contents: q (B:file1, 9, y)(C:file1, 4, y)

In this case, A is preferable to B, and A can add (B:file1, 9, y) to its history, but B cannot add anything.

mirror B: file1: contents: q (B:file1, 9, y)(C:file1, 4, n)

In this case, if we have no other information, then neither file is preferable (there is a conflict), and no histories change.

mirror B: file1: contents: q (B:file1, 9, y)(C:file1, 5, y)

This is impossible under normal operation if p≠q, since p and q both claim to equal C:file1@t=5.

mirror B: file1: contents: p (B:file1, 9, y)(C:file1, 5, y)

In this case (note p=p), B is the same as A, and A can add (B:file1, 9, y) and B can add (A:file1, 3, y).

If B was preferable to A as in the first three cases above, but the user marked a preference for version A, then B is free to add as shown, but A will update itself as if its scanned value had been new and that conflict had been resolved in its favor. In other words, any user-preferred value is always newer than the items made obsolete by the user making this choice.

If not choosing the algorithmically preferred value, then the value chosen by the user is considered to be an update over the algorithm's preferred value. More generally (allowing for algorithmic non-preference in the case of a conflict), the user-chosen value is given a history which is the max of the histories of the values considered by the user, and only issame when so in the max *and* so in the reality of the aspect value.

If the user marks a preference for version A, the history will become:

mirror A: file1: contents: p (A:file1, 4, y)(C:file1, 5/6, n)(B:file1, 9, n)

This could be done even if the files are not synced. [check how exactly the interface would allow this!] (It could also happen if the user tries to sync but the sync fails for some reason.)

The gossip spreading in these examples happens quite apart from any syncing or updating of actual files. It is just the spreading of existing history information from one actual file's history to another's. [show figure]

The info for a mirror in a history represents the last update of that mirror that this version of the file can remember. Info from new machines is only sent if the history on the target would be updated (in practice generally due to partial syncing patterns). A target is interested in gossip about versions that are known to be equal to its own or just plain obsolete compared to its own. It is not interested in versions that may be forking from its own.

Gossip can help resolve conflicts. Say machines A and B are synced to one version of a file and machines C and D are synced to a conflicting version. You sync A and C, specifying that A is better. Now if you sync B and D, they don't know that you have specified that A is better, and you will have to specify it again. But if you sync A and B first, then even though the file is fine (the same version is on A and B), A can send B the gossip that its version is better than the one adopted by C and D, and so then the sync between B and D will not exhibit a conflict.

A: file1: contents: p (A:file1, 1, y)(B:file1, 1, y)(C:file1, 2, y)
B: file1: contents: p (A:file1, 1, y)(B:file1, 1, y) + gossip(C:file1, 2, y)
D: file1: contents: q (C:file1, 1, y)(D:file1, 1, y)

Gossip is simply the propagation and incorporation of histories.

3.14 Guidance - Identification

Guidance (both identification guidance and preferred value guidance) is all the things that a human can indicate using the McSync interface:

- that a certain directory is the new version of a certain old directory
- that one version of a file is preferable to another
- that the permissions of a file should be set to 644
- etc.

Looking at a history and a scan (from each of several machines), a variety of heuristics are used to determine the identification guidance, which is simply the information about which scanned file is a continuation of which file in the history.

Since guidance includes the final heuristic of human go-ahead, it is treated as information from on high, on the basis of which final actions should be taken.

Recording more history for a file (due to changes noticed through the scan) depends on knowing which file in the history corresponds to which file in the scan.

A history is for a persistent file (a file tracking number) on a device. Scans present snapshots of files currently present on the device. Files can shuffle

around. So when we see a snapshot from a new scan, we have to figure out what persistent file (and thus implicitly what previous actual file) it corresponds to.

For example, if everything including the inode matches exactly, including ctime, between the history and the scan, then we can be positive it is the same file. But other cases are not so clear. In confusing cases, the user must help. If a file is copied to a new location and given a new name (something users often do when they are cleaning up their files), then perhaps the user will need to indicate that this should be considered as the continuation of the file.

McSync does not allow two files to both be considered as the continuation of a single file. Or, it does, but only through them both corresponding to the same virtual file, which happens through the virtual tree, not through the cp command. And even then they are not both continuations – they are distinct files with distinct tracking numbers, which happen to get synced in some or all aspects. If you really do duplicate a file or subtree and you want to keep it synced with (not forking from) the original, just add another graft to the virtual tree. Or for a file you could use a hardlink. If these aren't quite what you want, you might want a version control system, not a synchronizer.

When doing a scan, a list of all currently present files is made. This may bear a close resemblance to information contained in the history. The (often straightforward, but very delicate) task at this point is to figure out which McSync file tracking numbers on this device belong to which currently present files. Some new file tracking numbers may need to be created, and some old ones may be missing. The heuristics used can logically be anything, as the algorithm must be prepared for an arbitrary mapping of file tracking numbers to currently present files. But if they are easy to understand, that is good, so users can know what a nightly batch run will do.

If the inode is the same, then it is a continuation candidate. (On some filesystems (including macs), the inode has a birth date, which verifies whether it is the same or not.) If the parent and name are the same, then it is a candidate. ("Same" parent means new parent is a continuation candidate of the old parent.) If it is parallel to a parallel candidate on another storage device, then it is a candidate. (For example if two different files get renamed to the same name on different devices, then the remote predecessor becomes a candidate?) If there is more than one candidate, the user is asked to indicate the true continuation.

Between-file issues: There are some seemingly arbitrary restrictions on the mapping. These yield tree-based conflicts. If every file could have an arbitrary fully qualified path name independent of every other file, then this would correspond to the aspect/value abstraction. But there are restrictions between the filenames of different files: - a (existing) file's parent must exist - two (existing) files may not be in the same location - two files may not have names differing only in case on a case-insensitive system These restrictions also affect the order that updates must be attempted in, and ones that must be abandoned if a prerequisite fails, and caution must be exercised when deleting files on case-insensitive systems.

Once the mapping is chosen, it is represented as follows. Of course it is in reference to a particular history and scan. scanned file 1: file tracking number

72, scanned file 2: file tracking number 51, etc.

Different files on a single device might correspond to the same file in the virtual tree, but they still have distinct file tracking numbers and histories. A single file might correspond to multiple files in the virtual tree, but it still has a unique file tracking number and history. The virtual tree is just convenient notation for saying which files should be synced with which other files, and can change at any time at the whim of the user without affecting the syncing algorithm or any histories. Syncs can be between files on the same device just as easily as between files on different devices.

—- trying for perfect clarity....

A file tracking number, or tracked file, is different from a file on the disk.

A device has tracked files (recorded in its latest history), and actual files on the device. The actual files on the device (or their absence), as observed in a scan, is simply input for updating the tracked files. All algorithms (and most thinking about syncing) actually operate only on the tracked files, which are the continuous notion abstracting the ephemeral and ever-changing actual files, which in turn are in some sense there only when you go look at them. The purpose of the scan, achievable only with the help of identification guidance, is to update the tracked files so they bear a strong resemblance to current reality. A complete resemblance is unguaranteeable in multitasking unix-land.

Tracked files are similar to the files on disk, but they have all the extra history info needed for syncing. If we were designing the file system, tracked files could be the same thing as files on disk. But we're not. Each tracked file's aspect represents a syncable item. These are the items that can be synced.

Scans/identification guidance/noticing represent the attempt to get the tracked files to track changes that have occurred on disk. Instructions represent the attempt to get the facts on the disk to track changes in the tracked files. Everything else operates only on the tracked files.

We could say that the device sends its old history and scan, and the human sends back the new history, but the human cannot send back an arbitrary history, and it is useful to separate this process into identification guidance (the arbitrary human input) and noticing (a logical algorithmic step).

3.15 Guidance - Preferred Values

Users are allowed to specify all sorts of things in the interface. Some of it is in the form of identification guidance, saying which current files correspond to which historical files. But other things can be specified too, such as new attribute values, or preferences between different versions.

Preferance guidance is the user's indication of preferred values for aspects. (Not just names and parents, but all aspects.)

Preferance guidance takes the format of a scan, but also including file tracking numbers.

It is essentially a scan of the graft as it is in the user's head, where the location in the virtual tree serves to identify the file, rather than file path or

any other aspect. Maybe like inode?

If a user specifies in McSync that an older version is preferable over a newer version, then this is considered as a local edit, changing the file back to its former state. The older version is marked as obsolete, as is the newer version. Otherwise in later syncs other versions equal to or deriving from the new version could overwrite the version the user preferred. This is all because the algorithm treats increasing time as synonymous with increasing preferability. The user-specified version must be marked so as to be preferable to all known alternatives.

You should be able (in the McSync interface, but without connecting to other devices) to relocate a file to a location that doesn't exist on any local device, and the file will be removed, but the history will stay until this relocation propagates to other machines. If only the locations existing on the local device are synced, there would be a problem noticed at the time of resolving "between-file issues" like parent directory requirements.

This requires "unstorable" to be able to coexist with a specification of all the attributes. Perhaps only the large attributes (file contents and others where only a hash is stored) receive values of unstorable? No, "unstorable" and "notvisible" are extra information next to the ordinary information. "Missing" is the only one that can actually be a current value.

3.16 Noticing

Noticing is the act of updating a history with guidance and a scan. Once the guidance tells us which scanned file corresponds to which history file, we can update the history. We say the history notices the information.

Although the user manages the noticing on several machines simultaneously, conceptually the noticing happens on each machine independently.

The entry in the history for the current machine always gives the last noticed state of the file.

When the histories and scans from several machines are received, the user must be shown enough information to choose the correspondence between scan and history. We have the history from each machine, giving the updates it knows about from other machines. We have the scan from each machine, giving the current set of aspects on that machine. The user is shown a tree with every branch that is in any history or scan. But some branches are merged, because they have the same file tracking number.

If the file has ever been scanned before, then every aspect has been recorded for the local "mirror" (itself). Each aspect is examined to see if it has been changed, and if so, the adoption time for the local mirror is updated to now, and all other mirrors are marked as obsolete (not the same). set(post(h),t) = h + set($\emptyset$,t) If the aspect has not changed, then nothing in the history is changed for it.

If the file has never been scanned before, then it gets a file tracking number and is considered to be its only mirror and every aspect gets an adoption time of now. set($\emptyset$,t) = set(post($\emptyset$),t)

If A gets renamed to C on machine 1 and B gets renamed to C on machine 2, then this will be a conflict, but this specification of the continuations should be allowed.

If directories B and C get moved around so that A/B/C/D becomes A/C/B/D on machine 1 (not crazy – suppose B="2009" and C="receipts"), then we don't want to present the user with an infinite tree of B/C/B/C/..., even after the user has said that B and C correspond as they do. We cannot solve this by only showing paths that actually exist on some machine, since perhaps the correct tree does not correspond to any actual current machine state. The good solution is to show the virtual tree in the one way that it is supposed to be. Of course, a file present on some machine which is not supposed to be at that point in the virtual tree is marked as such, but if going through it further down the tree, you either jump to the appropriate point for that file in the virtual tree, or you are off the virtual tree and only looking at the machines having that file (the only way to examine files that need to be deleted). At each directory in the virtual tree, you see the virtual path, and all machine directories which correspond to that point. For the contents, you see all machine files which are in any machine directory corresponding to this point, as well as any machine files which belong in this directory. For each of these content files, you see the name under which it is supposed to be (there or elsewhere).

3.17 Instructions

When a comparison and/or discussion with the user have determined that a new value for an aspect is preferred over the old value, then instructions are given for an update. The instructions consist of the new value, and the new history for the new value. These instructions are sent to the target storage device, and hopefully soon executed.

If the update itself fails to be completed, but the instructions are there, then there are essentially two versions of the file on that machine. If these two versions conflict, then the update probably failed because the file was edited after the scan but before the sync. But if the instructions have a clearly preferable version, then the update may have failed due to permissions or access problems or due to a system crash or other form of abortion. This makes things more complicated for the interface, but it can allow failed syncs to propagate farther, if the interface treats the instructions as giving the correct version of the file.

The instructions can be thought of as another device, but a nameless one that doesn't appear in any history. The history accompanying the file is the gossip for the new file.

Of course if the file itself gets changed also, then there is a new conflict to resolve.

3.18 Syncing

A sync consists of instructions being followed.

Aspect values are changed either through external activity (user edits file, changes permissions, etc.) or through syncing. External activity is always considered to consist of purely preferable changes. Syncing only occurs when a change is determined to be preferable. So for a given actual file, all changes are increasingly preferable.

Even if a user overrode the default suggestion and chose an old version as preferable, the history accompanying the preferred version has been set to be strictly preferable to (post()) everything it should supersede. In general, the history is $\geq$ everything considered in choosing the preferred version. It is the sum of post /@ everything it doesn't match and id /@ at most one thing it does match (whichever version we say is the same, if any – but such a version must be $>$ all other versions).

When the sync occurs, A's aspect value is copied to B's, and if successful B updates its history (and so does A, in round 2). If B's aspect value (checked right before sync) no longer matches the one being overwritten (it has been changed even since the latest scan), then the update is cancelled and unsuccessful. If permissions or hardware failures block the copy, the update is unsuccessful. A device might retain the instructions and try again later, but it only succeeds when it succeeds.

If the update is successful, the histories are updated. The history at B is updated by taking all of A's history, plus B's history with everything marked to obsolete, plus B itself marked as equal since now, and using only the most recent record for each device. set(A + post(B), B.t) The history at A is updated (in round 2) by adding any of B's mirrors that are missing from A's mirror list, marking them as obsolete. B itself is marked as equal since now. The end result is the same at both A and B.

All of these changes are applied as one atomic changeset.

If A and B are not comparable (regarding preferability) then this is marked as a conflict. If the aspect value is nonetheless the same, then a sync occurs (of unspecified direction), and the histories are made to match by unioning the mirrors, keeping the info on the more recent update time at each (taking the max of the histories). Should they mark the update time as now, with the others (in max of histories) changed to obsolete? I don't think so. But perhaps I am wrong! (See Naive Merge Theorem.)

We have several types of sync:

- regular, non-conflicting (a newer version propagates over an older one)
- regular, non-conflicting (a newer version is the same as an older one)
- irregular, non-conflicting (incomparable versions are the same) (Naive Merge – record as new value)
- conflict, one version marked as preferred

- conflict, arbitrary value marked as preferred

In every case, when we do the sync (successfully change (or leave unchanged) a file aspect on a device), we update the history based on the histories of all devices considered when choosing the value. (The UI can let you say which were considered. Those are the versions you are saying the preference is preferable with respect to. This could be a superset of the versions you try to update right now.) These histories are merged to form a new history, by taking the most recent time T shown for device:file D:F in any history of this file aspect, and saying we are identical only if D:F was marked as identical in each history where it appeared with time T, and the preferred value matches this value. On the device where we are changing the value (or leaving it unchanged), we mark it as same since now (or since whenever it got this value). If the user wants the new value not to be considered the same as the identical most-recently-seen value on D, they can indicate this, and then issame is set to false for D.

While in principle, any two files on any machines can have an arbitrary aspect synced, McSync's virtual tree helps to insure that this doesn't get out of hand. Only files at the same point in the virtual tree are synced. And moving a file in the virtual tree (without actually moving the file) requires re-grafting. So it is natural for a fixed set of file positions (paths on machines) to continually be synced with each other. Files can of course be moved in and out of this set, by specifying a file's continuation as something in or out of the set, but this will in general affect (after syncing) all the files in the set, so it does not allow for information flow between stable sets.

The remainder is from an older version of this section: View of syncs It needs to be merged in or, more likely, thrown out.

When the workers update files, it is the same as if the user or someone else had updated the file. The success or failure of McSync's own update attempt is irrelevant. This update attempt can be viewed as being attempted by a separate process from McSync proper. What is relevant is the state of the file. When McSync scans the file again, we are hoping that it sees the preferred values. When it does, then Zing!, a sync occurs, meaning that two or more tracked file aspects have the same value at the same time. Actually, to be sure they had the same value at the same time, the HQ can ask for repeat scans of at least all but one mirror of the synced aspects. If we scan files A, B, C, and then D, and then A, B, C again, and they all give the same value, then they all had the same value at the same time (as far as McSync's histories can tell, which is all we care about), namely the time of the scan of D.

For safety, we need the fact that a tracked file has a single history. But this is practically the definition of a tracked file: its history. If there are (by mistake, say) two McSync directories both acting on the same set of OS files, then they each have their own histories for the files. There will be two tracked files, each with its own history, for a single OS file. If each one is surprised by the other's changes to the file, it just doesn't matter. From McSync's point of view, they are two files, being updated by who cares who. This could also happen from a single McSync directory if a disk is mounted in two separate places.

The lack of safety otherwise would come up in situations such as: We edit file X on machine B, and then run McSync on A, collecting scans of A and B. We tell it to propagate the new version to A, but we forget to hit "go", and then we edit file X on machine B some more, and sync B with C, so the new X propagates to C. We sync C with A, and we specify that actually A's old We later finish the sync from B to A Well, that example is a bit contorted, but the general idea should be clear.

Anyway, at the Zing! of the sync, the tracked file aspects are known perfectly to be identical, and the meaning of the sync is that they should be considered as in some sense the same; they should share their histories. If one of them is derived from an old version on machine A (currently unreachable), then any updates to any of them should propagate back to machine A the next time it is reachable. If any of them gets updated in the future, they all want to get that future update. These are assuming that the files continue to get synced with each other periodically, according to the default propagation behavior. Of course a tracked file can be released from this burden by de-grafting it. The synced files can be split into two sets with two different destinies by re-grafting them to two different places, not the same.

This is a nice view, but has some issues...

3.18.1 Updating a File

Suppose we need to change the contents of a file and change a permission bit. We should keep the inode the same out of respect for other systems like our own. I don't see any way to avoid having a window of vulnerability.

1. First we write a log file saying what we plan to do.
2. Then we get a file descriptor on the file. All operations will be done with the file descriptor.
3. Then we check whether we seem to have the right file. If any aspect has changed since the snapshot, abort due to being "in use" (and send back the new info so interface can report being in use and user can try again).
4. Then we change the file aspects we want to change.
5. Then we check to see if our change took hold.
6. Then we fix our history log to include the change.
7. Then we report back on any changes.

unix has only rare and unused/disabled mandatory locking. most locking depends on cooperation regarding the locking mechanism. which is no good for our purpose.

> If your only goal is to see if anyone currently has a file open at all, you can do things like "lsof" or "fuser" does... Eg: on a Linux system (and maybe others with "/proc"), you can just read through "/proc/¡pid¿/fd/", looking for the file... (Assuming you have permission to read through all the other processes' dirs, anyway...)

Chapter 4

McSync's Syncing Algorithm

4.1 Sequence of operations

Here we describe the sequence of operations (on the files and histories) involved in practice when syncing a directory tree across some devices.

0. CMD sets up a network of POs and WKRs and sets up HQ somewhere.

1. The histories of all the tracked files are collected from each device. (Each device loads the history and sends it to HQ.)

2. Gossip can be generated based just on the histories sent in. For any tracked aspect of any file, if $h1 \geq h2$, then $h1$ can update itself to $h1 + h2$. (HQ sends gossip to whoever could use it.)

3. A scan of all the files is collected from each device. (Each device performs a scan and sends it to HQ.) (Note that this step can start before the previous steps have finished.)

4. The user considers the suggested continuation candidates and specifies (tracked file, scanned file) pairs that go together, for each device. This includes pairs where one element is missing (i.e., new and deleted files). (Some scanned files or tracked files might get deliberately ignored for now by the user. No pairs are sent for these.)

These pairs are both kept at HQ and sent to the devices. They are used to update the histories as follows:

A (file, history) pair where the file's aspects match the latest version of the history does not need a history update. Any aspect that does not match gets its history updated to set(post(h),t). A scanned file with no corresponding history gets a history as above for a changed aspect, but not listing any other replicas, i.e. set(post($\emptyset$),t). A history with no corresponding scanned file is treated as if the value were scanned as "missing".

Since most of the pairs are saying there's no change, and these don't lead to any kind of update, what is the difference between them and a deliberately

ignored file? Nothing. Ignoring files simply means that they aren't noticed by this scan. If you put off figuring out the proper continuations until a later scan, that's ok. Scans aren't really part of the algorithm; they're just a useful heuristic for figuring out continuation guidance, and when that is acted on so as to extend the history, then that is the atomic action that feeds into the algorithm.

5. The user considers McSync's suggestions and chooses corresponding files (aspects) to be synced, with a preferred value for each aspect. This value might be the same version as in one or more members of the set being synced. Conceptually, this value is written to a temporary device, along with its history. This temporary device full of useful information is then sent to devices needing updating. We call the info on the temporary device "instructions".

The history of the aspect version on the device is the sum of the contributing histories. If there is a non-post-ed contributor, it must have the same aspect value.

6. Devices receiving instructions try to execute them.

If they fail, neither file nor history changes.

If they succeed, the history h from the temporary device is adopted (h should equal h + g, where g is the existing history), plus set(h,t).

Success is reported back to HQ, which similarly updates its history records.

7. HQ sends out gossip as in step 2.

Note that the temporary device needs to have been listed in the histories for the gossip to know that the same version was adopted everywhere, if this version is not one of the preexisting ones.

We should consider whether to do this. Can it be recorded elsewhere and still be robust to interruption? Probably not. It is only needed in cases where the preferred version is new (or an obsolete version, forcing it to be treated as a new version). Then a temporary device id needs to be created, just for these instructions going out (and others going out at the same time, perhaps). And at some point the temporary device needs to be deleted from all these histories. Basically, as soon as any update succeeds, and all devices hear about it (through gossip) (including the temporary one, in pending instructions), then the temporary device can be erased. Without the temporary device, successful updates present a naive merge situation.

Or better, McSync could let you set aspect values on any particular tracked file, same as using an editor or shell. Then, once the value you like is actually set somewhere, you can prefer it in the usual way. Or it could do all this automatically for your preferred value.

4.1.1 McSync operation

The user starts the TUI/GUI/batch commander (CMD).

CMD sets up the distributed communication network.

CMD creates a headquarters (HQ) on some machine.

HQ sets up workers as needed, including at the CMD device.

CMD can do whatever it wants. Here we will outline the typical case. HQ and workers are more deterministic.

CMD tells HQ to get scans (Ah) and history updates (AH) from workers.

To get a scan or history update, HQ tells worker what scans and histories it already has, and worker sends delta.

HQ analyzes results and sends CMD updates for whatever device directories CMD is tracking,

HQ also forwards any useful gossip (probably very little) back to other workers.

CMD may instruct HQ to forward scans or histories to other workers.

If CMD is collecting guidance from user, it saves it up.

When CMD is ready, it sends HQ pieces of identification guidance and preference guidance to act on.

HQ sends the identification guidance and preference guidance to the workers, who update the histories and files accordingly.

Any updated files yield new scans which the workers can add into the history, as no new identification guidance is needed for this.

The workers send in the new histories resulting from these new scans.

The HQ propagates histories as needed.

4.1.2 The anatomy of a sync

Upon connection, a device sends its stored histories (AH) in response to a request from the user/preferences/algorithm (UPA). (The UPA may be on two machines, one for UP and one for A, in which case to others the UPA seems to be at A.) The UPA typically also requests fresh scans of parts of the storage device. The results of these scans (Ah) are sent to the UPA.

Then the UPA can work on the identification guidance and preference guidance. Simultaneously, some initial gossip (aH) can be returned to the machines, which is only nontrivial if the web of syncs is a mess or if a new device was added. This initial gossip is just how the device's history can be updated based on all other received histories.

When the UPA is ready with the identification and preference guidance, it sends out identification guidance (AH) together with non-updating preference guidance (aH). It also sends out instructions (AH) for how to try to update the actual files.

Then the device tries to follow the instructions and update its files. It sends back records of its success (ah) to the UPA, which allows the UPA to know how the device's history has changed.

The UPA can then send final gossip (aH) about successful updates to other machines.

We need a little language, where h1 ¡– h2 means that history h2 is incorporated into history h1, and h1 h2 means that part of h2 which could be incorporated into h1, etc. Then we can write some pseudocode in that language.

4.1.3 Older Versions

The user might even use McSync to indicate a preferred aspect value that does not exist in any stored file. That's fine. Feedback is sent from the user to the devices: 1a. renamed and moved files and directories: which of our histories pertain to which currently scanned files (guidance) 1b. additional historical information from other storage devices involved in this scan (any new gossip) 2a. what is the preferred value of each aspect (guidance: gossip if value is ok, instructions if it is not) 2b. what history should accompany the new value (gossip) These are all received through two transmissions of a full history (separate for each storage device). The first contains the noticed continuation guidance and initial gossip. The second contains the preference guidance and instructions. The first is just information to fold into the history. The second has some of that (for aspects the user prefers as is) and some information to fold into the history only if we can successfully change the files on disk (for aspects that need to be updated (synced)).

For example, a file could hear through gossip that machine C adopted an outdated version at time 3, and then could hear through user preference that machine C adopted an outdated version at time 5. In this case there is very little point to the first piece of gossip, unless it is sent earlier than the human-confirmed items. The human-confirmed data can also fall into two categories, since identification guidance can indicate that a file has new contents and thus some other mirrors are now out of date, while the preference guidance can indicate that in fact the file should be synced if possible to a better version, with a different history.

Then, what do we, the storage device, do? We need to update our histories, and update the stored aspects. To do this in a recoverable way, we go aspect by aspect. 1. We update the stored aspect to match the master version (if there has been no change since the scan). (This is the "sync"!) 2. We update our histories according to the success of (1) as well as any further history in the master version. This completes the first round, and then round 2 is done, which is almost the same as round 1, but the "scan" consists only of the success for updated files (for which we know what tracked file they correspond to). In round 2 the user can see what failed in round 1. Also, we find out in the round 2 feedback what succeeded on other machines, allowing us to improve our histories even further.

At the central machine, it is: Scan in, guidance out, sync, success in, guidance out.

The algorithm thread is in charge of everything. The TUI thread is like an advisor to the algorithm thread (and the -batch option essentially substitutes the TUI with a trivial default-follower). The algorithm thread reads the specs file to configure itself initially (although the -wait option will prevent automatic actions like automatic starting of remote scans), and then it proceeds to do its things, notifying the TUI of developments during each stage.

Each device stores each history and scan sent to each other machine (delete when machine says it got new one ok).

A machine can send file version N. If it hears back that the other machine has file version N

4.1.4 Two Rounds

Round 2 is done automatically because (I) at the end of it the state of all participating machines is stable (further rounds would have no effect, unlike at the end of round 1), (II) round 2 can be performed much more efficiently than simply repeating round 1 since no new scans need to be done or transferred and no user input is needed, and (III) the extra information distributed in round 2 can help resolve conflicts. An example is given below (Example 1).

The equivalence of round 2 to round 1 means that for analysis of safety, correctness, etc., it is enough to consider repeated occurrences of round 1.

Example 1. How the second round helps synchronization.

Say A and B are laptops that you use, and C is a machine at the main office that you can reliably reach because it is always turned on and reachable with a predictable ip address, as opposed to your other laptop.

Say you edit file F on machine A. Machines A and C are then synchronized. Note that without round 2, machine A does not know that C now has the new version.

You also edit F on machine B, and then you synchronize B and C. The two changes properly show up as a conflict, and you resolve the conflict by saying that C should adopt the new version from B.

Later we are back at machine A and synchronize with C again. This same conflict will show up as a conflict again, even though you already resolved it!

[I don't think this is right – machine C knows its version is preferable to A's.]

If we had done round 2, then machine A would know that machine C adopted its version at time T, and that the new version was adopted by C at a time > T, so this tells us that machine C already somehow figured out that the new version is to be preferred, and this tells us how to resolve the conflict.

4.2 Data Structures

What is actually stored in histories, instructions, etc.?

We have 4 file formats:

Ah. Aspect values, but no histories.

This is like a truck full of untagged boxes.

Sent in by a scan.

AH. Aspect values with histories.

This is like a truck full of tagged boxes, where the tag points to a history record.

Sent in by a connected machine. (Its pre-scan local knowledge – probably largely known to receiver, compressible.)

Sent out after collecting identification guidance (noticing). (End of history matches current state.)

That one doesn't seem to need the aspect values, just the tracking number.

Sent out as instructions for update-inducing preference guidance. (End of history indicates desired changes.)

ah. No aspect values, no histories, just a success bit.

Needs to be with reference to a particular stack of tags.

Sent in after attempting to follow instructions.

aH. Histories without aspect values.

This is like a stack of tag records.

Initial gossip (like when a machine hears that there is a new device which it matches).

Non-updating preference guidance (when you indicate that an aspect's value is actually preferable to other devices).

Final gossip of new histories based on successful syncs.

4.3 Formalized Algorithms

Algorithm 1. How a scan is merged with a previous history to make a current history. (Specifically, how McSync decides what changes are name changes.)

See "guidance" section above.

Algorithm 2. How histories from various machines are compared.

If two sources claim to have no deviations from a common ancestor, they are the same. (If they are not actually the same, flag an error!) If a source claims to deviate from an ancestor, then it is preferable to that ancestor (and to any source matching the ancestor). With this information we can reduce the number of locally most preferable versions. If they are all the same (for example there is only one, but also for matching files on first sync), then there is no conflict, and necessary propagations can proceed. Specifically, the trivial sync is "performed", and the histories are updated to be at least as preferable as any of the histories. Otherwise there is a conflict.

Algorithm 0. How to detect cyclic grafts.

First, what is meant by a cyclic graft is that on some device, a directory and a subdirectory are mapped to the same point in the virtual tree. One can see that this is if-and-only-if with the fixed-point of syncing being an infinite tree, which is why we consider it to be a problem. It is also if-and-only-if with a directory being mapped to two points on the virtual tree one of which is a subdirectory of the other. The point is that a directory should not be specified to contain itself.

Example 1. If we graft a/b/c onto x/y and a/b/c/d/e onto x/y/d, then we have trouble, because a/b/c/d = x/y/d = a/b/c/d/e.

Example 2. If we graft a/b/c onto x/y and a/b/c/d/e onto x/y/e, then we have no trouble. We have a/b/c/e = a/b/c/d/e.

Example 3. If we graft device1:a/b/c onto x/y and onto p/q/r device2:d/e/f onto p/q and onto x/y/z then a/b/c = p/q/r = d/e/f/r = x/y/z/r = a/b/c/z/r

= trouble.

If we prefix every path by the name of the device or virtual tree, then our question is whether any string is equivalent to a strict prefix of itself, with grafts giving word equivalences. The fact that every graft involves the virtual tree is irrelevant, as we can use a disjoint directory of the virtual tree to identify two device paths usings two grafts, which is then equivalent to a single graft between devices. The fact that there are multiple devices is irrelevant, as we can imagine the device names to be sibling directories in a single tree. Thus the problem is like the word problem for semigroups, except that all strings in the productions start with a special start symbol, and the question is whether any string is equivalent to a strict prefix of itself.

Another wording of the problem is, given a set of bidirectional stack rewriting rules, is any word's equivalence class infinite?

The difference between examples 1 and 2 shows that the analysis cannot just be on the topology of the tree – it can matter whether the symbols being used in different parts of the tree are the same or not.

—— puzzle (here to end of section)

Given a set of prefix equivalences on binary strings, efficiently determine whether any equivalence class is infinite.

(prefix equivalence: 10 = 0 means any string 10xyz is equivalent to 0xyz, etc.)

Example:

10 = 0, 1010 = 01 —- answer: infinite: 101 = 01 = 1010, so 101 can pump out 0s

10 = 0, 1010 = 00 —- answer: finite

If the equivalence applies everywhere in the string (not just at the beginning), then the general problem of whether two words are equal is unsolvable (like the halting problem) (it is the semigroup word problem). But always being at the beginning of the string should make it much easier, and the finite/infinite question is somewhat different, too.

It just came up in a program I was writing. Roughly, suppose that when you mount a filesystem, it doesn't obscure the files that were already there, but instead it copies all the files both ways so both systems have all the files. You could then mount a third filesystem in some subdirectory of that mount point, and then all the files there would be duplicated on all three filesystems. The only danger is that this process could lead to an infinite number of files, say if you mount comp1:/foo/bar onto /bar and comp1:/foo/bar/foo/bar onto /bar/foo, because then the file comp1:/foo/bar/foo/tmp would spread via the first mount to /bar/foo/tmp, and from there the second mount would spread it to comp1:/foo/bar/foo/bar/tmp, and from there the process would repeat with "bar/tmp" instead of "tmp", so it would become bar/bar/tmp, bar/bar/bar/tmp, and so on. This is exactly the first example I gave for the puzzle.

Random comment:

This problem is the same as considering a nondeterministic reversible stack machine that doesn't have a separate input channel, but rather takes its initial

stack as the input. The question is to determine, given a machine, whether any stack can grow without bound.

This problem is also the same as considering a nondeterministic reversible Turing machine which always erases when it moves to the left, and starts with a blank tape on the right. The question is whether the left half of the tape can be filled so that the machine fills the right half.

Theorem:

Any large equivalence class implies that there is a prefix P in one of the given equivalences such that the string P can be transformed into a longer string having P as a prefix. (P can pump something out.)

Proof:

Consider equivalent strings A and B, with B much longer than A. Consider a sequence T of transformations (prefix substitutions using the given equivalences) that takes A to B.

Each of at least the first —B— - —A— characters must have been produced by an element of T. (All characters in a transformed prefix are considered to have been produced by that transformation, regardless of any similarity to the replaced prefix.) We associate each such produced character of B with the element of T that produced it.

Since the number of characters that can be produced by a single transformation is bounded by the given prefix sizes, the characters of B must be associated with many members of T. Indeed, there must be two such elements of T that performed the same prefix substitution.

Note that whenever some characters are associated with an element of T, then the entire initial segment of B, up to those characters, can be derived from just the changed prefix of that element of T, acted upon by all ensuing elements of T. (All remaining characters of B were produced by earlier elements of T, or were present to begin with, so they are untouched by later elements of T.)

Considering the two elements of T that performed the same prefix substitution, we see that a prefix of the first such element led (via the subsequence of T between the two elements) to a longer string where that same prefix again appeared. Q.E.D.

Algorithm:

We simply need to expand the given equivalence classes as far as possible, using the given equivalences. If we ever produce a prefix string that has a prefix that is in the same equivalence class, then it is a pumping prefix, and we have an infinite equivalence class. If no such prefix string can be produced, then by the theorem there is a limit to the size difference of different strings in an equivalence class, so the process will terminate. This means we must produce strings in a way that will produce all strings that can be produced, even if there are an infinite number of them (e.g. breadth-first search instead of depth-first).

Treat the given input as a graph with the prefixes as vertices and the equivalences as undirected edges. Furthermore, whenever one prefix is a prefix of another prefix, we draw a directed edge from the longer string to the shorter string, labeled with the excess characters in the longer string.

Then we add edges in the following way: If two vertices both point to equivalent vertices (in the same component, considering only undirected edges), then we compare their edge labels, and add an undirected edge (between the source vertices) if they are equal, or a directed edge if one is a prefix of the other, labeled again with the excess characters.

(Note that the original edges could also be obtained by using this rule for adding edges, if a different set of original edges were used: Each vertex, instead of being labeled with a prefix, could have an original arrow going to a global extra vertex, labeled with the prefix.)

The new edge (directed or undirected) is annotated with the two edges that made it, to make it easy to find the actual sequence of transformations corresponding to the edge. If it is an undirected edge, the merging of the two components makes a note of the vertices between which the edge was formed, and path compression is not performed in the union-find. This way, a component can give a path between any two vertices in terms of paths in its subcomponents.

The meaning of a directed edge is that the source, minus the label (which is always a suffix of the source prefix), is equivalent to the target. In other words, the source is equivalent to the target plus the label.

The meaning of a vertex is that it corresponds to a state, the state of having a base that can take that prefix. An undirected edge means the same base can take the other prefix as well. A directed edge means we can lengthen the base (with the edge label) and then it can take the other prefix as well.

Might we try to add a second directed edge (with a different label) between two vertices? Unclear, but if we do, then we have an equivalence between two different prefixes of the source string, meaning that the shorter one can pump itself into the longer one, generating an infinite equivalence class. These edges would be labeled with AB and B, meaning that pref is already equivalent (through the target component, between the target vertices) to prefA. Thus this multi-out-edge issue does not need to be checked for if the destination component has already been checked for cycles and prefix-induced edges.

Since we never add vertices, this process of adding edges must end. If at any point we get a semi-directed loop (a direction-respecting cycle containing at least one directed edge), then this cycle can pump up a string indefinitely.

If there is a sequence of transformations by which a prefix can pump something out (as guaranteed by the theorem in the event of an infinite equivalence class), then this sequence must show up as a semi-directed cycle in the graph, since every possible sequence of transformations is embodied in the graph. (See diagram.)

Complexity:

equivalence component $\longrightarrow$ incoming edge tree, subcomponents, connection vertices of subcomponents vertex $\longrightarrow$ equivalence root, rank edge $\longrightarrow$ source, dest

Create all prefix vertices in a tree, put arrows on queue. Add equivalences as undirected edges. Nothing needs to be checked for yet.

Now start processing arrow queue.

First check if arrow is alredy there (use dest comp's incoming edge tree, or check components for undirected case).

To test whether adding a directed edge leads to a cycle, we do depth-first search (DFS) (of components) backwards from the edge's source, looking for the dest.

To test for other edges to add to the queue because of the directed edge we are adding (by the add-edges rule), we compare the new edge to the existing incoming edges for the dest component.

To test whether adding an undirected edge leads to a cycle, we do DFS both ways to find and report any problem path.

To test for other edges to add because of the undirected edge we are adding (by the add-edges rule), we compare (and merge) the two trees of incoming edges, finding all pairs that are super/sub prefixes of each other. The edge is added simply by unioning the components, storing the vertices of the edge.

Example:
10 = 0
10x0 = 01 // really two examples, x=0 or x=1

10x0 $\xrightarrow{x0}$ 10 // two initial arrows
01 $\xrightarrow{1}$ 0

if x=1, then 10x0 $\xrightarrow{0}$ 01 // two examples diverge here
// if x=1, then we put an arrow on an undirected edge, yielding a cycle!

CSE = DEF
D = QRS
Q = AB
C = ABR
then DEF = QRSEF = ABRSEF = CSEF = DEFF

4.4 Proofs of Correctness

Consider a partial order "preferable" on versions of the value. (A version being a value that the user sets the aspect to and is then propagated here and there. If the user changes it and changes it back, that is a new version of the value.) We define a value Z to be "immediately preferable" to a value Y if (A) the user changes Y to Z, or (B) the user uses McSync to automatically or manually decide that Z should propagate over Y. We define "preferable" as the transitive closure of "immediately preferable". We define $Z > Y$ as "Z is preferable to Y".

You can think of the entries in the history struct as giving the most recent time, for each machine, that it adopted a value Y such that we know our current value Z is $\geq$ Y, and it also tells us whether $Z > Y$ or $Z = Y$.

Two lemmas:

Lemma 1:

Note that for all values X from earlier times (for a given tracked file) than Y's time, we also know that our current value Z is $> X$, since $Z >= Y > X$. We know $Y \neq X$ because when we say "adopted a value Y", we mean that Y is considered new (not matching the previous value) at that point.

Lemma 2:

For all values X from later times, the given tracked file yields no information on the relation between Z and X. We could have $Z < X$, $Z = X$, $Z > X$, or $Z \lneqq X$ (conflict).

Thm:

One history A is preferable to another B if, for every machine, one of the following holds:

- A has issame=NO and B has issame=NO.

Old: Both versions feel superior to some old version on that machine. Not so informative.

- A has issame=NO and B has issame=YES and $t_A < t_B$. This was ok before, but not for the new user-change approach.

Old: This just means that B, while obsolete, has more recent information about that machine than the newer A does.

New: This means there was a user change on that machine that A hasn't heard about.

- A has issame=NO and B has issame=YES and $t_A \geq t_B$.
- A has issame=YES and B has issame=YES for $t_A > t_B$.

and for some machine, the second or third holds.

Machine by machine, compare the histories of the two versions V and W. Each machine casts a vote.

For machine A:

	$t_V < t_W$	$t_V = t_W$	$t_V > t_W$
$V = W =$	$V < W$	$V = W$	$V > W$
$V = W \neq$	$V < W$	$V < W$	$V > W*$
$V \neq W =$	$V < W*$	$V > W$	$V > W$
$V \neq W \neq$	$V < W*$	no opinion	$V > W*$

The third column is computable from the first.
The third row is computable from the second (7 cases).
Non-starred ones are decisive.
The starred ones used to be no opinion.
There is always a non-* (decisive) vote, and now the *s must match it.

Then the votes are tallied. Any machine voting "no opinion" is ignored. The remaining votes must all match, or else there is an error (kind of like a conflict, but shouldn't even happen). If there are no remaining votes, that is a conflict. Otherwise, the unanimous decision holds, and inequality leads to automatic propagation.

Upon automatic propagation, the written-over history is changed to obsolete on every machine, the machines being written to are changed to the current time with equality, and the max of the histories is taken and used on every machine being written to.

If there is a conflict but the values are the same, we do a naive merge, updating each machine's record to the latest knowledge.

But what if one version knows about various conflict resolutions (on unique machines) between versions that are all known to be obsolete? In this case, those conflict resolutions need to have not been marked as user-changes. This is possible if those were conflicts where every machine voted "no opinion",

Ok, what if there are more than two versions... ?

max(h1,h2) means that for every aspect and tracked file in any of the histories, we take the max over the various histories (each of which has at most one entry for that aspect and tracked file), using the increasing order:

[absent history] (works like (t=0, issame=NO))
(t=1, issame=YES)
(t=1, issame=NO)
(t=2, issame=YES)
(t=2, issame=NO)
...

However, any time with YES should in fact be the max; the presence of a higher value is an error.

h1 ¡– h2 means the action of incorporating history h2 into history h1. This means that we

h1 h2 means that part of h2 which could be incorporated into h1

The past light cone of a file goes up the tracked file lines, and backwards across the sync arrows (which are usually bidirectional). It is red at first, and then any time it crosses a change (X on the line), it turns blue.

Q1. Can red and blue meet? Can we get purple?

A1. The only merging point as you go up is at the source of an arrow. Purple is prevented here by next paragraph (*).

This red/blue structure is the history. The black and white picture (tracked file lines and sync arrows) is the common global truth, while the red/blue structure is a particular line's view of it. The history data structure only records the lowest colored change on each tracked file line, and whether it is red or blue, but that is because that turns out to be all that is needed for the algorithms. We can think of the history as the whole red/blue structure emanating from the bottom of a given line.

(**) What is the consistency relation, if any, between different histories? (This is what we would need to check that various sync scenarios don't violate.) (1) For the black/white part (doesn't belong to any particular history), the consistency criterion is that no history pic from anywhere has any purple. You can't go around an X on a line by using sync arrows and other lines without hitting another X somewhere. (2) Different histories must be consistent with (derive from) a single black+white picture. We will show below that we don't allow the addition of arrows that would make purple, and purple is thus avoided. No single history ever gets purple. The histories do not imply a unique B+W picture (two syncs in a row are no different from one, etc.) but they are consistent with at least one (the true one, assuming McSync has been working correctly) and

they are not consistent with any picture that allows a purple history, if every individual history is purple-free. What if a history is absent? Then maybe if we were to add it, it might have purple. Say we just have one history, and all kinds of crazy syncs and change points on lines that that history doesn't even touch? But come on, every line has a history, no matter how short. And B+W is defined by matching it.

The point of avoiding purple is what? That we will always have one of the following three cases? That we can achieve a sound partial order on all stages of all files? Probably the latter, since that is used for guarantees of preferability.

Comparing two (or more) histories means we draw them both (all).

1. If the red parts touch, then the files are the same.

2. Otherwise, if one red part touches any (necessarily blue) part of the other, then it is the obsolete one. This means that prior to syncing, it must get a new change point (even if the aspect appears to stay the same), reducing the red part to a trivially small one. In other words, (*) a sync is not permitted if red and blue overlap. In practice, this is the interesting case of automatic propagation: it gets a change point by changing it to match the other. If each red touches the other blue, they will both need a change point.

3. If the red parts each do not touch the other history, then the files are incomparable. If the aspects are the same, no change is needed before syncing, and we call this a "naive merge". Otherwise, a preferred value needs to be chosen ("resolving the conflict") simply so that the files can be the same, a prerequisite for a sync. Choosing a preferred value (and executing the instructions to change the value to the preferred one) is simply a way of editing the file.

4. Say one red part touches both red and blue of the other (must be on different lines, otherwise both reds change to blue at the same point). This means the first red part (A) shares a same ancestor with (B) at (X) and also at (Y), but the only change points along the circuit (A)-(X)-(B)-(Y)-(A) are on (B)-(Y). Since syncs are guaranteed to be only for identical files, we know that the change points on (B)-(Y) result in a net change of no change, but that is ok; even a single change can have this property. Say identical files start out on (X) and (Y). They propagate via some 2-way syncs eventually both reaching (A), where the final sync reports no conflict only because the files already match. They also propagate on other machines towards (B), where again they match, even though the (Y)-(B) path contains a change point. (B) is unaware of most of the syncs towards (A), seeing at most the first syncs away from (X) and (Y), and happily syncs with no new change points required. Now we have a cycle of guaranteed-equal files jumping over a change point. Let's use it to create a problem. Let's say the (Y)-(B) path had two change points between which the file was different, and this difference propagates towards (C). Then we change (A) to match (C) so they can sync. Now (A) and (B) are each better than the other, so they conflict, even though an omniscient observer would prefer (B).

If (X) and (Y) had derived from a common ancestor, the the sync at (B) would have forced a change point on (X)-(B), giving (B) ¿ (A), and the partial order is preserved in the sensible way, with (B) being the most modern version.

So if we do a sync, then maybe the red parts overlap or maybe they don't, and same for blue, but red and blue do not overlap with each other.

By doing the sync, we are adding arrows between the synced lines, at the bottom, in the guaranteed-red part of each line. (Or near the bottom, at the guaranteed-same-contents point in time.) This enlarges each red region to be the union of both red regions, or more generally, for directed arrows, enlarges the destination red region by adding in the source red region, and enlarges the destination blue region by adding in the source blue region. It is basically just the union of the two pictures.

sync = link = bond = tie = coupling

Thm:

This unioning of the pictures (even n-way) is the max operation defined above.

More precisely, the history struct of the union of the pics is given by the max of the history structs of the pics.

Pf:

Any line containing both red and blue has the same change point dividing red from blue in every picture that is colored at that point, because in every picture, each line is colored from some source arrow (or from the bottom) on up, so if there were two different red/blue borders, the lower one would send blue into the upper one, which is disallowed by (2).

So, any line containing red in any picture will record the change point (first scan time below it – change points are hats on snapshots anyway; no arrow can intervene, and arrows are guaranteed to be at places where the snapshots above and below (or at) are identical) where red turns blue, since red cannot pass through any other change point. This is the lowest change point on that line in any picture.

And any line containing only blue will also take on the lowest blue change point of any picture.

Regarding change points being hats on snapshots (scans)...

Def:

We can define the tracked file as being something quite different from the OS file. The tracked file is like some kind of backup, which only changes when a scan is done, only changing if the snapshot differs from the previous one. Snapshots can also be marked arbitrarily as change points, at the time when they are appended to the line.

Syncs are only done between tracked files.

Regarding the data format:

Thm:

The data structure is rich enough.

Pf:

The only operation we need the histories for is the comparison, yielding case 1/2/3 above. This simply needs detection of when the red part touches the other red, the other blue, or none of the other, on a given line.

Red touches other red if $t_a = t_b$, and both have issame=YES.

Red (A) touches other (B) blue if $t_a \leq t_b$ and t_a has issame=YES and t_b has issame=NO.

Red also touches other blue if $t_a < t_b$, both having issame=YES.

Red doesn't touch other if $t_a > t_b$ or t_b not there.

This covers all possible cases from total order, and the total order is sufficient for determining which case holds.

Thm:

Change points all represent desirable changes.

Pf:

Induction & transitivity.

4.4.1 Lossless Theorem

Any change (by a user, not an automatic propagation) to a file cannot disappear unless the file is "further modified", meaning that the further modification is a change made to a version that already reflects the original change. (If you are worried that the further modified file might then be able to disappear, just apply this theorem to it.) Note that the further modification may have already occurred in the past in some other location, unknown to you.

This theorem holds even if the past histories of files are a little screwy, due to "conflicting" space-like separated decisions, like A getting preferred over B in one place while B gets preferred over A in another, so that the partial order breaks down.

If you are really worried that there is a competing version that might trample your version, tell McSync to mark the file content (or whatever aspect you fear for) as updated. This just cancels the issame flag for all but the current location and thus a conflict will be triggered if any competing version comes along.

Proof: Let's look at a machine that makes the change. The change can only be erased on that machine by the existence of a sequence of values (from other machines) starting with the value on that machine (when it was on other machines), meaning further modifications occurred on other machines.

Restatement: A can only propagate over B if a user has determined that A is preferable (perhaps transitively) to B.

Proof: We will inductively assume the theorem holds for previous changes. We also assume that any change occurring outside McSync has the property that a user has determined the change to be preferable. In McSync, information Z is only propagated (over other information Y) in a given image if it can be verified that in some image (let's call it the "progressing" image), Z was adopted at a later time than Y. In particular, Y has to have been adopted at some point in the progressing image. So by induction we know that the changes in the progressing image leading from Y to Z were each determined as favorable by a user, and so by transitivity, Z is similarly preferable to Y, and so the theorem holds for the propagation of Z over Y in the given image.

4.4.2 Optimality Theorem

A will automatically propagate over B whenever it can be determined, from any conceivable set of stored information within the past light cone of the machines being synchronized, that (i) A is preferable to B, and (ii) B is not preferable to A.

Note that condition (ii) is just for safety's sake; in practice it only occasionally comes into play.

Note that this theorem, together with the definition of "preferable", is practically the definition of the algorithm.

Definition: The past light cone is determined by histories being transferred between devices. McSync transfers these whenever you let it. It can happen with or without a sync. If there is a sync, some more history will be transferred (if possible) if the sync is successful, so the other devices know it was successful.

Proof:

We will consider a naturally optimal algorithm which has full information about the past light cone, and then show that the data structure we use is actually sufficient for doing the same thing.

Full information would mean that we know (1) every time the file was changed by the user, and (2) every time a file was changed by a sync, and (3) every time a user told McSync that one file was preferable to another. Of course, we can only know this within the past light cone.

For (1), we really only know about changes at times when the file was scanned by McSync.

For (2), we know that the new file is preferable to the old files used in that sync, except for the old files it is equal to.

For (3), you could imagine that a user resolved a bunch of conflicts but wanted to postpone the file transfers to a later time.

4.4.3 Consistency Theorem

$h1 \sim h2$ At most one of $=$, $<$, or $>$ holds among all $(h1.t, h2.t)$ for tracked files present in both histories.

The algorithm always maintains $h1$ $h2$.

4.4.4 Local Improvement Theorem

The ancestor sequence for a file aspect on a given device has the property that each child is preferable to its parent. "This version is the best this device has ever seen."

4.4.5 Global Improvement Theorem

The set of ancestors (on different devices) stored for a file aspect are all less/equally preferable than/to current version on this device. "This version is the best that

this device has heard of."

4.4.6 Naive Merge Theorem

If A and B are not comparable (regarding preferability) then this is marked as a conflict. If the aspect value is nonetheless the same, then a sync occurs (of unspecified direction), and the histories are made to match by unioning, keeping the info on the more recent update time at each. Can this "naive merge" cause any problem?

Yes it can! Here is a simple 6-machine example.

Say we use two machines A and B, and every so often we sync them with random other machines (think USB sticks) but not with each other directly. In year 1, we make the directory of interest be unwritable (on both machines – chmod is easier than mcsync, and McSync can certainly understand that a twice-made change is at least as desirable as a once-made one!), and we sync A with An, and B with Bn. Then in year 2, we change the directory to be writable, and sync A with Ay, and B with By. Then in year 3 on both machines we change the directory to be unwritable again. Then we sync A with Bn, and B with An. These each result in a naive merge. Then we sync A with By, and B with Ay. These each result in an update to writability! The machines all agree on the wrong value! This is clearly the fault of how the naive merge is wreaking havoc with the space-time structure.

The naive merge needs to be less naive. It needs to produce a history that won't let the latest change be erased.

Say the histories only record user changes. McSync-changes that are worry-free applications of the algorithm (automatic updates) do not get their adoption time on that machine recorded, although any user change sources of the obsolete version are set dirty. So the histories are simply smaller. Any two histories that would otherwise have been comparable through this machine will still be comparable, since strict preferability is the prerequisite for the automatic update to proceed, and this is what is needed at that point for the transitive conclusion to follow.

The only point of an issame flag is to help us to be obsolete. It allows anything that is marked as being later on that machine to overwrite us. If we mark automatic updates on A with an issame flag, then we have to allow overwrites by versions for which we only know that they are better than a prior version on A (clearly on some other machine they must be clearly preferable, since the histories on A in this case aren't very enlightening). However, if we only use the issame flag for user changes, then we can prohibit overwrites by such versions, as they are clearly unaware of something important that happened.

Now change points are only marked at user changes, and the rules for preferability need to be modified.

Like, say we only change the directory to be unwritable again by resolving the conflict in favor of unwritability. Should we have the original unwritability of the other machine as a source? Then we will get overwritten by its intermediate value upon sync with By. This is bad if in fact our own value was better, as

in the example above. Should we say the orginial unwritability of the other machine is obsolete? Then we won't get overwritten by its intermediate value upon sync with By. This is bad if By is B's final value, and the sync with Bn represented an accurate merging of identical mirrors. Also, if we say that, then we should say both values are obsolete, and then we get needless conflicts.

So, the only way to resolve this that I can see is rather ugly: Have a global time. That is, record the real times of snapshots, and ensure that on a single storage device (which may be connected to different computers/clocks at different times), the snapshot times are always increasing. Then, we can warn (like a conflict) any time there would be an automatic update to an "earlier" version. This can only happen as a result of a naive merge. CHECK

—

4. Say one red part touches both red and blue of the other (must be on different lines, otherwise both reds change to blue at the same point). This means the first red part (A) shares a same ancestor with (B) at (X) and also at (Y), but the only change points along the circuit (A)-(X)-(B)-(Y)-(A) are on (B)-(Y). Since syncs are guaranteed to be only for identical files, we know that the change points on (B)-(Y) result in a net change of no change, but that is ok; even a single change can have this property. Say identical files start out on (X) and (Y). They propagate via some 2-way syncs eventually both reaching (A), where the final sync reports no conflict only because the files already match. They also propagate on other machines towards (B), where again they match, even though the (Y)-(B) path contains a change point. (B) is unaware of most of the syncs towards (A), seeing at most the first syncs away from (X) and (Y), and happily syncs with no new change points required. Now we have a cycle of guaranteed-equal files jumping over a change point. Let's use it to create a problem. Let's say the (Y)-(B) path had two change points between which the file was different, and this difference propagates towards (C). Then we change (A) to match (C) so they can sync. Now (A) and (B) are each better than the other, so they conflict, even though an omniscient observer would prefer (B).

If (X) and (Y) had derived from a common ancestor, the the sync at (B) would have forced a change point on (X)-(B), giving (B) ¿ (A), and the partial order is preserved in the sensible way, with (B) being the most modern version.

We can see an example of this with computers A,B and USB sticks C,D,E lying around, all keeping track of the writable flag for a directory. The following sequence of events is perfectly legitimate. With McSync, you should be able to plug any of your USB sticks into any machine at any time and sync the two.

```
        A                       B                       C                       D
t=1     n: A1 B1 C1 D1 E1       n: A1 B1 C1 D1 E1       n: A1 B1 C1 D1 E1       n: A1 B
   user
t=2     y: A2 b1 c1 d1 e1       y: a1 B2 c1 d1 e1
   B->E
t=3                             y: a1 B2 c1 d1 E3
   A->D
t=4     y: A2 b1 c1 D4 e1                                                       y: A2 b
```

```
   user
t=5     n: A5 b1 c1 d4 e1
   A->C
t=6     n: A5 b1 C6 d4 e1                                  n: A5 b1 C6 d4 e1
   user
t=7     y: A7 b1 c6 d4 e1
   A<>E
t=8     y: A7 B2 c6 d4 E3
   B<>D
t=9                              y: A2 B2 c1 D4 E3                                  y: A2 B2 c1 D4 E3
   B<-C
t=10                             n: A5 B10 C6 d4 e3      n: A5 B10 C6 d4 e3
```

Now machines A and B are incomparable: A+C prefer A, while B+E prefer B. This appears to be resolvable because B's version can trace itself to A:t=5 while A's version can trace itself to A:t=7 and B:t=2, so the most recent user action has to have been either A:t=7 or B:t=2. If B merges with F, and F was set to n by the user, then B can also trace itself to F:t=2, and for all we know this was more recent than the user's activity on A, so we cannot resolve it.

—

Each user action can be bounded in time between two scans. These are distinguished from McSync actions, which are typically bounded between two scans during the update process that are very close in time (the first to check that the file hasn't changed, and the second to check that it has). However, a McSync action that was specifically requested by the user is considered a user action.

We might want to resolve conflicts of actively incomparable files by considering the most recent user action of each. The most recent user action can be bounded in an interval. Merging intervals for a single file (as when syncing identical versions of it) is done by taking the max of the interval starts and the max of the interval ends: We know the most recent of all the user actions is in this interval. When intervals are compared, they are only comparable if they do not overlap: If they overlap, we do not know which user action was more recent.

Now, do we want to resolve conflicts in this way? If the user had updated B at time 10 in the example above (with the final B-C sync at t=11), the final histories would be exactly the same, but the most recent user action would then favor B, not A. And indeed, it would feel like the right thing to favor B in that case, although a conflict is also defendable.

Why can't we just resolve all conflicts in this way?

If the same file was created independently on two machines, then there is no reason the more recent one would be better.

- What if we can't find an old text file, so we start trying to recreate it, and then on another machine we find it and move it to the right place, so then at next sync there is a conflict. Actually in this case it is the filenames that conflict, and the more recent action is the correct one, but still it shouldn't

automatically delete the new one.

- Suppose we start editing a large text file on one machine, and then also on another machine (not noticing that we hadn't synced in between). When we do the sync, we don't want to lose anything.

—

Going back, can we resolve these in a symbolic way? Suppose for issame values we have yes, no, and source, where source is a form of yes that indicates that that tracked file was in fact a (the) source of the current value.

When histories are extended with snapshots, change points are marked as "source" points if they seem to have been changed by the user, not by McSync.

Then our final states are:

```
       y: §7 §2 ≠6 ≠4 =3       n: §5 =10 =6 ≠4 ≠3              a clear win for A: A
and if at t=10 the user had changed B, then after the B-C sync at t=11 we would have:
       y: §7 §2 ≠6 ≠4 =3       n: §5 §10 =6 ≠4 ≠3              a clear conflict: A w
while if at t=2 the user set F to n, and then at step 9.5 merged B with F, we would hav
       y: §7 §2 ≠6 ≠4 =3 ≠1    n: §5 =10 =6 ≠4 ≠3 §2           a clear conflict: A w
```

The idea is that (not only can the McSync interface tell the user where changes originated but) a file must have a source. Consider the potential results of the two directions of propagation for the first of the cases above, the "clear win for A".

```
A->B        y: §7 =11 ≠6 ≠4 ≠3        y: §7 =11 ≠6 ≠4 ≠3
```

or

```
A<-B        n: =11 =10 ≠6 ≠4 ≠3        n: =11 =10 ≠6 ≠4 ≠3
```

Of these two, the first is clearly much better. If both results have sources, then there is a conflict (if the values are the same, then both results are the same). If neither result has a source, that is probably impossible, because probably we can prove something like the latest source can't disappear. Yes, that would make for a nice theorem.

Thm:

The latest source can't disappear.

Pf:

A source can only disappear if something is preferable to it. To appear inferior, it would have to meet an item (somewhere) which is $\geq$ its times for the issame machines (and strictly ¿ if it is issame in the other history too). The source is an issame machine. For it, strictly ¿ is not going to be present in the other item unless the source machine itself already lost itself as a source, which it didn't by induction. So what about $\geq$ with the same time? This could happen if it meets a preferable item with a different value. Again killed by induction: The $\geq$ can only arise by meeting something which is preferable and therefore already has $\geq$.

But wait! A source can naively merge with something else, which is elsewhere hopelessly obsolete, and then the source is in danger.

Thm:

If a result has no sources, then its value has already been user-selected against directly downstream of every location where it was user-selected for.

Pf:

Consider a place where it was user-selected for. At that place, something else turned out to be preferable later, which is difficult, like proof of previous theorem, essentially requiring that the machine or something directly downstream meets a user action.

Claim:

If both results have sources, then there should be a conflict. Justification:

Otherwise sources can disappear, particularly the latest source could disappear. Unless one source is definitely not the latest. But if one source were determinably (given the full space-time structure) earlier than another, then there would be a path going forwards from the earlier one to the later one, in which case the later one would have canceled the earlier one.

Claim:

If just one result has sources, then nothing is lost by jettisoning the other. Justification:

We know that the sourceless result had all sources user-selected against, i.e. it is an obsolete value in any event. The sourceful result on the other hand is definitely user-downstream of the sourceless result. That is a good enough reason right there!

—

The problem with putting change points at naive merges is that then if we have unsynced identical copies on A,B,C,D we sync A-B, C-D, and then B-C, then

If one version is preferable to another, but the values are the same, we still mark the unpreferred one as getting a change point (all values not issame) to represent it quickly passing through whatever sequence of events yielded the other version.

—

Another idea: Upon merging of equal incomparables, keep track of the two (or more) histories you would get by one of them overwriting the other. Then see if this list of histories is adequate for resolving preferability issues. If so, see if the history list can be compressed, say to a max and min value for each machine.

But then what do you do when one history says C should prevail, and the other says there should be a conflict?

Redo steps 7,8,9 above.

—

We could say the point is to avoid cycles of equalities that break the partial order.

—

4.4.7 Ghost Deletion Theorem

Proof that info about ghosts can safely be deleted once all known mirrors agree with (have performed) the deletion.

(show that info can't have any effect after that anyway)

(have a way for a system to know that gossip has been to all corners of the world and back)

This is safe because when we hear about a mirror, we also hear about everything it knows about. So when we hear that the file has been deleted by every mirror, then we also know that we have heard about every mirror any of them knew about at the time of the deletion. Any later mirror is irrelevant anyway, since it already mirrors the deletion of the file. In other words, we know we have what was at some space-like time the full component, and the file was gone everywhere.

Could a device hear back about its own history of a deleted file for which it has deleted the history? It would seem so, since history deletions happen at different times on different devices, so after it gets deleted on A, B could still decide that the file had come back to life somehow.

4.4.8 Free Preferability Theorem

If a user specifies in McSync that an older version is preferable over a newer version, then the older version is marked as identical and the newer version is marked as obsolete. Other versions are marked as obsolete unless their aspect value matches the older one (even if not marked as identical to it).

Prove that this does not lead to any inconsistencies – somehow it is interpretable in a meaningful way.

No, it is BAD!!! The only way to show that the reverted state is preferable to the newer version is to mark it as an even newer version. Marking it as equal to an old version invites other copies of the newer version that may be floating around to overwrite the preferred reversion.

Any change on a machine must be accompanied by a strict increase (with regard to preferability) in the history for that machine. There is no such thing as a reversion. This is not a versioning system.

Any sync is accompanied by taking the max of all contributing histories, even where nothing changes. The synced value can be anything (needn't even match any machine – free preferability!), but any machine where the sync changes the values must advance to a strictly greater history. To allow this, the sync must have a time which is greater than the scan time. (The scan may be not so recent and may have propagated elsewhere, so we cannot pretend the scan saw the new value.) During the sync, McSync checks to see if the sync worked. This is a scan.

Say the scan right before the sync finds that the file changed, and in fact it changed to be what the sync wanted it to be. Then the sync can report that it is surprised, but it should mark the sync as successful – it is the same as if the scan had been requested (as just a scan) by the TUI, and the user had seen the

preferred value of the file, and we assume they would still have done the sync, which is in this case a trivial sync (no aspects need to be updated), affecting only histories.

4.4.9 Arbitrary Distinguishability Theorem

issame can be set to false on any individual item in any individual history. Note that this can lead to a conflict when there would otherwise be none. But that is probably what was wanted, if anything. We just want to prove that anything that wouldn't have been overwritten won't be overwritten this way either. Well, we could have had a conflict due to incomparability that this action in fact resolves. Again, that would seem to be what the user intended, if anything. Actually, I don't see how that could happen. All I see is that a conflict, of the sort where two differing files have histories claiming they should be identical, could be resolved by setting issame to false on one of the files.

4.4.10 inode and ctime theorem

A brief commentary on using inode info (from stat()) to detect changes in a file. Summary: If inode# and ctime are same as last time, then nothing has changed since then, unless user is mucking with the clock.

ctime gives latest change to file or inode apart from changes to atime due to reading the file. man 2 stat says ctime is: Time when file status was last changed (inode data modification). Changed by the chmod(2), chown(2), link(2), mknod(2), rename(2), unlink(2), utimes(2) and write(2) system calls.

Renames (mv) change the ctime on linux but not on mac. This seems to be at odds with the man 2 stat page.

Since atime only seems to be manipulable by also writing mtime, ctime will record the time of any deliberate tweaks to atime, but writes to atime based on genuine file access do not update ctime.

The file could be different if the ctime is the same, because files on a mac may have been shuffled around. Checking the inode as well seems failsafe: If the inode is the same but the file is "different", it means the file was deleted and then another was created that got the same inode. But then the ctime will necessarily be newer than the old ctime. Of course, if the person is futzing with their clock, or the clock is low resolution, then all hope is lost. Presumably the new nanosecond fields for these times are to allow a specification that the clock always advances between disk writes or disk reads.

btime of course simplifies this: wherever we were thinking of "inode", we can use "(inode,btime)" as an identifier that is never reused.

4.5 Security

A slave starts by sending its version number, signed by the home office. The master also sends its signed version number. If slave needs to update, then it

moves itself, downloads and compiles a signed new version, and transfers control. If master needs to update, it moves itself, downloads and compiles signed new version, and alerts user. There is no security concern because an uncompromised system will only download and compile code that has been properly signed by the home office.

4.6 Efficiency

If a file has not been modified, its contents do not have to be reexamined. If a subtree has not been modified, then its unchanged history does not need to be transmitted. Certain themes in the histories recur in many files; these are compressed.

4.7 Discussion

4.7.1 General comments regarding synchronization in general

Inaction can never be detected. If a wise person checked a file (or its permission bits or whatever) for accuracy (and found it accurate, so did not change it) while a fool changed it on another machine, the wise person's inaction cannot, in any scheme, conflict with or override the fool's action (unless every update is marked as a potential conflict, in which case you have a modification detector, not a synchronizer).

Is there any way for the wise person to "positively take a null action"? Yes. She can mark the checked material as having an adoption time of now on the current machine. Then if anybody simultaneously changes it elsewhere, it will be detected as a conflict. However, if her inaction first propagates to the fool's machine, and then the fool changes it, there is no way for McSync to know not to trust the fool, so this is not a very reliable method. Better is to mark (in the McSync interface, not as part of this algorithm) the fool's machine as dubious or the file as needing approval.

Similarly, mistakes cannot be detected. If you delete a file by mistake, this deletion will be propagated. If you are used to CVS's behavior where you can delete a file to get the repository's version, you should be aware that synchronization is not like this – there is no repository! If you want to use McSync to get a deleted file back from another machine (assuming the deletion has not been propagated to the other machine yet), you will have to use the interface to mark the older existing version of the file on the other machine as preferable to the newer deletion.

If a normally-mounted NFS partition is not mounted for some reason, it can look like the whole partition was deleted. Such a deletion can be propagated. To avoid this, you can (again in the McSync interface, not in the algorithm) (A) mark the mount point as an "intermittent" place where if it's missing it should be treated as ignored rather than as deleted, or (B) set a

deletion limit – if more than that many files are scheduled for deletion, then no action is taken. (getmntinfo gives info like df to detect this in darwin, http://cboard.cprogramming.com/linux-programming/114785-mac-os-x-10-4-getattrlist-know-file-system-type.html .)

Synchronized file systems are no different from any other file system: You should not give a fool access to your data. You should take care not to delete important data. A synchronization solution is not a backup solution, and vice versa.

To partially help deal with this sort of situation, the McSync interface does allow you to mark some or all machines as dubious for any given path. Then if new values on that path arrive from a dubious machine then propagation will occur only if manually approved.

4.7.2 Quandaries

What if one machine has a file A that is out of sync with B on the other machines. Then you delete A. Should the other machines delete their B? Case 1: A is out of sync because it has been modified since the last sync. Answer 1: Then the deletion is part of the modification McSync will propagate. Yes. Case 2: A has been modified since before the last sync, but was not fully propagated. Answer 2: Again, the deletion is preferable to the modified form which is preferable to the old form. Yes. Case 3: A is an old relic, and all the other machines have the newer version B. Answer 3: This is a conflict, because A took a different trajectory than B, considered from their common ancestor.

Say you have a directory for big files, that is only duplicated on a subset of machines. Say you move a file there. Then, on a small machine not duplicating that area, the file is removed. This is correct. The history log on the small machine should mark it as "unstorable", which is extra information in addition to the aspect value which remains in the log. Say you then synchronize this small machine with a big machine that has not yet heard about the move. The big machine will in fact complete the move successfully, because the information in the history is enough to know how to move the file. Then, if you sync the second big machine with the original machine, what happens? The machines are actually in perfect agreement. The only problem is if an "unstorable" aspect is large (like file contents) and changed – then only the hash is known to the intermediate machine. Then the target machine needs to find contents with the given hash before it can execute the instructions. Not so bad! (Maybe you prune almost everything from your handheld, and never want to sync the big stuff across it).

What if I have a directory that has the contents of a remote ftp site I maintain, where the remote computer only grafts that directory. My main home directory includes that directory. What if I move a file out of the ftp directory and into my home directory? Then the remote ftp machine should record "unstorable" (not "missing") in its history and delete the file. It needs to record the motion of the file in terms of the virtual tree? No, the virtual tree is just a shorthand for what real locations correspond to each other. The new

real location may correspond to multiple positions in the virtual tree. Then it should be synced to all the real locations those virtual locations correspond to (and away from all the real locations it used to correspond to). This is no different from an ordinary file move.

Suppose a virtual directory has a photos, movies, and poems subdirectory, and on each real machine these three are grafted to three individual places whose relative location does not match on the different machines. One day you notice that one of the files in the photos directory is a poem, so you move it. If a machine has the photos but not the poems, it will simply delete. If a machine has the poems but not the photos, it will simply add. If one of these machines is a bridge to a machine with all three, the machine across the bridge can do things correctly if the added/deleted poem's history is correctly stored on the intermediate machine, which it should be if the intermediate machine had/has any tracked file who would hear the history. Say you are at a machine with just photos when you see the poem. You want to delete it, but you know that on other machines it has a place, in the poems directory. What should you do? You should start up McSync, and relocate it with McSync. You can even do this locally, without connecting to any other machine, and the saved guidance and history will propagate the change to other machines when a sync occurs later, and in the meantime the poem is already gone from that machine.

And what machine should be marked as having adopted an update that is requested by the McSync user? No machine. Suppose the user indicates that the file should be writable by world but the file is owned by root and the user doesn't have that kind of privilege. Then this update will fail everywhere. If it were to succeed in one place (the user's machine at home), that one place should not list anywhere else as ever having adopted this change. Only scans/guidance and successful syncs lead to updating the history.

Say the user specifies the continuation of A as B. Say the syncs fail (somewhere). Is the specification of the continuation lost? No, it is still there, in some guidance. Even if the sync succeeded somewhere, and there is a round 2, the change will not be noted on a device where the change fails, since that device essentially didn't participate in anything. Could there be a virtual storage device, where syncs always succeed? Doesn't make much sense in the context of distributed syncs between neighbors, unless there is one virtual device per device. Yes, that is what the instructions provide. The virtual device does correspond to actual storage in a way, in the history file itself on that device. Permission or name or location changes could then propagate through machines that fail to make the change. However, if the change is failing, it might be a bad change! Not worth designing mcsync around a misfeature!

What we want is that if we say a bunch of files should be moved to a place mirrored on A from a place that wasn't, and the copy fails because say the disk is full, then we don't want to have to select all those files again. The next time we fire up McSync, it should know that those need to be copied to A. This is separate from any syncing algorithm, since it falls into the heuristic side. Nevertheless, this knowledge should exist on any system that could possibly know about it. So what is the form of this knowledge? It is part of the knowledge

sent to the remote machines after the user reviews the suggested changes. So it is verified info about file continuations and preferred values of aspects. Let's call this "identification guidance" and "preferance guidance". It is information direct from on high (the user) about what the right thing to do is.

Say A=B=C=D.

`A: a1=, b1=, c1=, d1=  B: a1=, b1=, c1=, d1=  C: a1=, b1=, c1=, d1=  D: a1=, b1=, c1=, d1=`

Then we modify A and C and sync A=B and C=D.

`A: a2=, b2=, c1≠, d1≠  B: a2=, b2=, c1≠, d1≠  C: a1≠, b1≠, c2=, d2=  D: a1≠, b1≠, c2=,`

Now we sync A and C, saying we prefer A. And we sync B and D, saying we prefer D.

`A: a2=, b2=, c3= d2≠  B: a2≠, b3=, c2=, d2=  C: a2=, b2=, c3=, d2≠  D: a2≠, b3=, c2=, d2=`

Now we sync A and B. How is the conflict noticed?

Well, B thinks it's more recent than A, according to machines A and B. But A thinks it's more recent than B, according to machines C and D. So there is a conflict.

Say we have a ring of machines, and the left half are synced with version X and the right half are synced with version Y, and X and Y are in conflict. Syncs happen between pairs of machines adjacent on the ring. At the north boundary, Y is marked as preferred, and at the south boundary, X is marked as preferred. The boundary goes counterclockwise, while the knowledge about the preference also goes clockwise. When the knowledge hits the boundary, then the mismatch triggers a conflict, which gets resolved one way or the other. Say on the other side the conflict is left unresolved, because everything is run in batch mode on that side. When the mismatch is resolved, the newly preferred value is marked as being more recent than either older value, although equal in practice (but not through issame) to one of them. This preference propagates in both directions around the circle, changing everything to the new preference. It can pass through the unresolved border in either direction.

What about case sensitive file systems vs. case insensitive ones? On any particular case-insensitive device, files after the first one (by tracking number? by dibs? better be by dibs) will be marked as "unstorable". The user can change filenames to resolve the conflict in the interface, or can tell McSync to use an alternate filename (`r_eadm_e7.txt`) on this device. Alternate filenames might also be useful in general, though it's hard to see why. Alternate locations (in the directory tree) are handled instead through the pruning/grafting mechanism.

How should the info in the specs file (describing grafts on the virtual tree, storage devices, ip addresses, source code, etc.) be propagated? Same as everything else? It could just be its own branch on the virtual tree, with the preference to be synced automatically with any system we connect to.

What if a file on a device gets regrafted to a different point on the virtual tree? What do we do with its history? Well, the history actually doesn't care about the virtual tree, and only makes references to actual files on actual devices, so the history can remain. The virtual tree is just a guide as to what should be synced with what. In fact, any file can be synced with any other at any

time. That is why the virtual tree hardly gets mentioned when the algorithm is discussed. Of course the history is only useful when the same files get synced over and over.

What if we want to use a small USB stick to keep two large machines synchronized, and the changes at any given sync can fit on the USB stick, although the full static contents of the large machines cannot? Can we use McSync to do this? The USB stick would have to hold the scans and histories of the large machines, and it would hold a copy of any file that is in flux, perhaps in its directory of file contents having certain signatures. In general, scans, histories, guidance, and instructions should be carryable on a third device. As another example, if there is a network of machines that are updating files and syncing with each other automatically, and there is a subset of the network that you can reach while using the McSync interface, then a conflict elsewhere in the network should be resolvable by the user. For example if the user syncs A and B, while B and C sync automatically, then a conflict between B and C should be resolvable from A, even though A matches B. This will work if McSync has been set up (say the automatic sync is run from C) so that C's scan also gets saved on B, and A can fetch C's scan from B. In general, the algorithm can be run on a different machine from the interface (say the interface is run on a PDA with very low bandwidth, but the algorithm should be run on the office machine that has high-bandwidth connections to other big machines all storing much more than the PDA), so we need to distinguish the algorithm machine from the interface machine. The interface machine is where McSync is initiated by the user. Scans are automatically sent from reached devices to the algorithm device, and can optionally be sent (or retrieved from, more automatic) elsewhere too.

How are hardlinks dealt with? What if two hardlinked files become separate files? A hardlink is like a forced sync on many but not all of the file's aspects, while the files themselves are not mapped to each other as "identical" in the sense of McSync's unique file tracking number for each file on the device. If at any time we can sync any two file aspects, then there doesn't seem to be a problem – any sync involving a hardlinked aspect is actually also syncing with the hardlinked aspects of the "other" files (whose hardlinked aspects are conveniently stored in the identical location on disk). McSync's histories simply record things as if the hardlinked files are always getting synced whenever "needed", with the actual implementation of such a sync being done in part by the filesystem itself. We can't stop that sync from happening, so we just record it.

How are softlinks dealt with? Suppose l1 -¿ /usr3/cook on device 1 and -¿ /Users/cook on device 2, and on device 1 we change l1 to /usr3/cook/tmp while on device 2 we move tmp to temp. Then we expect l1 -¿ /usr3/cook/temp on device 1 and -¿ /Users/cook/temp on device 2. What is getting synced in this case? The link has syncable contents IFF the target files have tracking numbers? The full or relative path of the target is getting synced. Elsewhere we don't deal with "..". See "Symbolic links" above.

4.7.3 multi-paths

[Consider the alternative] McSync allows you to say that two files are both the continuation of a single file. (To a certain extent – they won't have the same file tracking number.) They simply each get the history of the previous file, although at least one will clearly have a new parent or name. Then either of these files can be synced with a remote version of the old file, but depending on which one you sync, the remote file will move to one of the two locations. The other location will probably get a new file with matching history except that every aspect has an adoption time of now on the remote machine, which can lead to spurious conflicts. Any edits to remote versions will be applied to both local versions at the first opportunity, upon which the remote version will also be split. I don't see the benefit of this. If you want to branch into two versions, you want a version control system. If you simply create a new graft in McSync, and let the files be duplicated to the new location by McSync, then they will have the appropriate histories. This will not split remote files. This seems like a good thing. Now consider merging of histories. McSync lets you say that a single file is the continuation of two different files. For example, maybe you notice that you have two versions of the same file. You could specify that remote versions of these files should both sync with the single local descendent, which would imply them moving into the same position, but an edit to either of the remote files would be applied to the local file. You might as well just delete one, if there was an edit it will show up as a conflict anyway. Even better, sync the two files (using some kind of "merge" feature in the interface, which shouldn't require any dedicated support in the algorithm). Then McSync's syncing algorithm will combine their independent histories in the right way. If there would be some kind of conflict with the two files moving to a single location on some machine, that will be brought up as a conflict. If merges and splits are allowed, what if two files merge on A while one of them splits on B? Seems ok. But I still don't see when this is useful. It could certainly add confusion. So let's not implement it until we see why it would be useful. It is like splitting a file into two versions, but the split takes effect on each device at the time that that device first hears of the split, rather than at any concrete time. Very strange. And anyway, the parent directory and name are aspects of a file. If two actual files correspond to the same point in the virtual tree, the only way this can be consistent is if they are mapped there with two separate grafts, so some prefix of their path can differ. Ok, this consideration is about the idea of allowing multiple parents, which might be what hardlinks have anyway. But I'm still against it. [End consideration]

4.7.4 Beyond ASCII

This is relevant to us for filenames, which we need to display. This requires knowing the character encoding used by the device, and the character encoding used by the terminal (for the tui, anyway).

UTF-8 seems to be very nicely usable. Characters 0-127 are the same as

ascii, 128-191 are continuations of multi-byte characters, and 192-255 initiate multi-byte characters (although only some such multi-byte sequences are valid UTF-8).

UTF-8 is meant to replace ASCII in the future, so at some point "text file" is going to mean "UTF-8 file" just as it means "ASCII file" now.

One issue is that different characters are different numbers of columns wide. Some far east character sets use two columns per character, and some accenting modifier characters occupy zero columns. This is directly relevant to screen formatting.

Another issue is that there are different encodings. UTF-8 is one, but some places are likely to keep using more specific encodings for their local character sets, since that can be 1.5, 2, or 3 times more efficient than UTF-8. Do these get used in filenames? In terminals? I don't know.

Another issue is that there are multiple ways to represent accented characters, due to the flexibility of unicode. From the rsync FAQ: An example of the latter can occur with HFS+ on Mac OS X: if you copy a directory with a file that has a UTF-8 character sequence in it, say a 2-byte umlaut-u (
0303
274 or 11000011 10111100 = UC 00fc), the file will get that character stored by the filesystem using 3 bytes (
0165
0314
0210 or 01110101 11001100 10001000 = UC 75 0308, where 0308 is "combining diaeresis") the above Mac OS X problem would be dealt with by using –iconv=UTF-8,UTF8-MAC (UTF8-MAC is a pseudo-charset recognized by Mac OS X iconv in which all characters are decomposed).

The issue for us is with filenames on disk, and with screen display (and keyboard input) in the terminal window. Does a filesystem have a locale? Yes, but everyone who's talking about it is moving to UTF-8, except windows, which is UTF-16. A terminal also does.

How do Hebrew and Arabic users type in and store their filenames? It looks like we need to reverse things for screen formatting.

xattr attribute names are UTF-8.

```
man wcswidth
man mbstowcs
man wgetc, WEOF
http://www.delorie.com/gnu/docs/glibc/libc_89.html
http://www.cl.cam.ac.uk/~mgk25/unicode.html
  #include <locale.h>
  int main()
  {
    if (!setlocale(LC_CTYPE, "")) {
      fprintf(stderr, "Can't set the specified locale! "
              "Check LANG, LC_CTYPE, LC_ALL.\n");
      return 1;
```

```
    }
    printf("%ls\n", L"Schöne Grüße");
    return 0;
}
```

4.7.5 Extended Attributes

man 2 stat the basic stuff

man listxattr stuff in addition to stat (selinux will hide attrs that you aren't allowed to see – oh well)

list/get/set/remove xattr

foo/..namedfork/rsrc

macs seem to store unstorable xattrs in the ._ file

to read the com.apple.ResourceFork, use NULL pointer to get full size, the read chunks using pointer arg

There's no intrinsic difference between the Finder info you get back via FSGetCatalogInfo[Bulk] and the Finder info you get back by "com.apple.FinderInfo". The only gotcha is that the BSD routine returns the data big endian, and the File Manager returns the data native endian.

There's no hard and fast limit but the practical limit for the size of a resource fork is just over 16 MB.

In my testing of xattrs, I believe the limit is around 3800 bytes.

Resource forks can be much bigger, bigger than we would like any other extended attribute to be. Note that getxattr has a special parameter, position, that's specifically present to allow you to read the resource fork in chunks.

Oh, yeah, and if the resource fork is to be stored in an AppleDouble file, it must be under 4 GB.

The limit for other extended attributes is also not well defined. In fact, its file system specific. In general, if you're storing more than 1 KB in an extended attribute, you're in the weeds.

`http://lists.apple.com/archives/Filesystem-dev/2008/Feb/msg00014.html`

How can I tell that the file is a resource fork instead of some random file named `._xxx`? I guess check the xattr?

In Mac OS X 10.4 and later, you have the choice of using either File Manager or BSD. In the BSD case, you should access all of this stuff via the BSD extended attributes API. In the File Manager case, you can can use either FIle Manager or the BSD extended attributes API to get at the Finder info and resource fork, but you must use the BSD extended attributes API to get at other extended attributes.

The situation in Mac OS X 10.5 is mostly unchanged. The only new wrinkle is that, if you're working at the BSD level and you want to access the resource fork as a stream (as opposed to an extended attribute), you can now do so (by appending "..namedfork/rsrc" to the path).

In all cases, you should ignore AppleDouble files (those whose name begin with "`._`") if you encounter them in the file system.

—

Under OS X the mandatory data and resource forks of a Macintosh file are exposed on HFS+ volumes as `filename` and `filename/..namedfork/rsrc` to BSD environments and as filename and `._filename` when "split" (as for use on foreign file systems.)

Traditionally the Macintosh stored certain metadata regarding files in the Desktop Database. Under Mac OS X the `.DS_Store` file associated with a directory contains similar information such as directory background info, the position or order of files and their icons within a directory, and more. When copying a directory you should probably consider copying these.

Creation date vs btime? Are they different?

`http://archive.netbsd.se/?ml=samba-technical&a=2006-08&t=2304036hascodeforos-indepe`

Mac and Linux have getxattr(), fgetxattr(), listxattr(), flistxattr(), removexattr(), fremovexattr(), setxattr(), and fsetxattr(), and Linux has lgetxattr, etc., to not follow links. The arguments on the systems are different.

```
convmv -r -f ISO-8859-1 -t UTF-8 --notest *
```

mmc: Check out Carbon Copy Cloner: Improvements to Backup Bouncer

`http://www.bombich.com/groups/ccc/wiki/7ba51/Improvements_to_Backup_Bouncer.html`

"I'm happy to report that rsync 3.0.7 passes an extended version of the backup bouncer test suite with flying colors."

mmc: getting all the advantages seems to require that the demon at the remote end is also the new version.

4.8 Comparison with Other synchronizers

Comparisons of

optimistic (non-blocking, lets things get out of sync, if trouble ensues then roll back to safe spot or in this case ask human for help),

peer to peer (decentralized),

non-immediate (do it when you want)

synchronizers:

What is it:

Name, source institution, year made, platforms, languages, interface, user base.

Efficiency:

Ease of installation, speed efficiency, space efficiency.

Functionality:

2-way vs. N-way, can add new machines, can start with existing copies, can do partial updates,

can have partial datasets, works fine with mv, cp, scp,

can recognize name changes, can recognize restructuring, completeness of conflict resolution,

robustness to interference,

efficiency of algorithm, scalability, wandering storage.

coda – client/server approach, like cvs

ficus rumor roam

unison

bayou – partial datasets not ok

panasync – version stamps

practi – needs to be built into the os, but provides greater flexibility in only exposing consistent sets of files

jefferson82 "Fast Concurrent Simulation Using the Time Warp Mechanism, Part I: Local Control"

Leslie Lamport. Time, clocks and the ordering of events in a distributed system. Communications of the ACM, 21(7):558–565, July 1978.

Tra: "File Synchronization with Vector Time Pairs"

Has an extra version vector that records syncs (as opposed to changes). This is the light cone surface. Syncs are one-way events.

3.2: How can it be meaningful to compare times from different systems? Is there a global clock accessible by all systems?

3.5.1: What does monotically increasing have to do with anything else? Assumes a sync of a large file system is an atomic operation.

Par. 2 presents a convoluted way of saying that there is typically a very high redundancy in the list of sync times and so it can be greatly compressed.

Par. 3: The lack of deletion notice sounds risky. For example, if only part of the directory was synced, then the relationship between the directory and its contents is unclear.

3.5.2: A5 is less powerful than A5,B3 because it is harder to compare with C1,B2. Just because anybody having A5 will also have B3, doesn't mean that systems without A5 have no use for seeing the B info. Par. 3: Directories have no info of their own?

tra, even if sync is done in both directions, seems to miss second round

csync2 – uses file aspects (stored in sqlite db!), "dirty" bit

chronosync Qdea's Synchronize Pro `http://www.decimus.net/` synk `http://www.syncsort.com`

`http://blog.plasticsfuture.org/2006/03/05/the-state-of-backup-and-cloning-tools-under-mac-os-`

4 dangers of research: might eventually turn out to be: wrong, trivial, irrelevant, or done before

Relation to previous work:

Version vectors keep track of an integer for each replica, which increments when that replica is modified. Then two files can be compared by comparing these vectors, and if they are comparable, the lower one gets updated to the higher one (incomparable = conflict).

This approach cannot handle the case where new replicas get created. Say a file gets replicated to a new machine. ?

First of all, we allow the integer version number to increase by more than one, so in fact it represents (either roughly or exactly) a time (local to that machine) when the file was observed to have changed, which may help the user in determining the value of the change when resolving a conflict.

Secondly, a given machine keeps track of an "is same" flag for each other machine to indicate whether or not the last known version on the other machine is the same as the current version on this machine. This flag is what allows conflict detection to work despite possible differences in the set of machines included in the histories of the two versions being compared.

Chapter 5

How Syncing is Conceptualized for the User

The user starts up McSync. It shows them the virtual tree.

At each point in the tree, we show the old/new(+where) aspect values, and when each device(+isreachable) was last checked.

In general, circumstances existing at a point in the tree should be indicated also higher in the tree, to make them findable.

Some circumstances:

- file has a conflict
- file has been updated on machine x
- range of dates when files were last scanned on each machine, or at least whether it is before/after starting McSync (or in-program reset point, if it is just always left running)
- total number of files, file sizes, storage space (per machine)
- a desired change is predicted to fail (disk not big enough, bad permissions, etc.)
- graft points, prune points
- virtual file corresponds only to a single device (due to grafts/prunes, i.e. no syncing will ever take place)
- file synced to other files on same device
- number of hardlinks greater than one
- renames / deletes / any other hardlink activity
- changed inode

- subtree preference settings
- files that "have no master version", i.e., every machine will update the file somehow. These files would be conflicts if aspects were not treated independently.
- unstorable, notvisible, missing
- file with different name on some device
- where various preferences are used
- where gossip will be spread

preferences can be given for any point in the tree and apply from there on down

preferences exist for:

- heuristics for guidance
- allow user to say that all changes at or below a certain point (perhaps coming from a certain machine) do (or do not) need to be manually approved, or should be overwritten (restored to previous state)
- Check mounted filesystems and mark any mount points in the graft so we know not to delete if missing (can McSync see if a directory is mounted?)
- what to do with case collisions on case-insensitive systems
- what to do with extra file parts (forks) on single-fork filesystems

5.1 Virtual Tree

From the point of view of the algorithm so far (the "history algorithms"), the virtual tree is simply a heuristic for helping users keep an orderly understanding of what should sync with what. From the user's point of view, the virtual tree is the real thing, with parts of it stored here and there on different systems. All file manipulations by the user occur through the virtual tree interface, which is one of the two foundational ideas of McSync.

There is a virtual tree of files. This is a conceptual tree holding all the files that are ever synced. Actual files must exist on an actual storage device, and grafts give a mapping between these actual files and positions in the virtual tree. Syncing occurs when multiple actual files are mapped (via grafts) to the same place on the virtual tree.

The virtual tree is somewhat nebulous, in that its structure beyond the graft points and prune points is given just by the contents of the storage devices, which are typically somewhat out of sync with each other.

Almost all McSync preferences/configurations apply to places/portions of the virtual tree. (The main exception is networking information.) Even the

McSyncDeviceArchive directory could be indicated by being grafted to /McSync, but that's probably a bad idea because it would encourage new users to poke around in it.

5.1.1 How the virtual tree lets you specify continuation and preference guidance

The user is shown the virtual tree. In a given virtual directory, we see: * all the tracked files which are grafted there (maybe the real file has just moved elsewhere) * all the snapshots which are grafted there (maybe the corresponding tracked file (to be confirmed by user) is still elsewhere) * Each filename is just shown once, and further details are shown if we select that file. * Further details show the snapshot state on various devices, tracked state on various devices, and the preferred state. * Name and Location are aspects along with the others. * You can mark snapshots as belonging to tracked files on their device, here or elsewhere, new or old. * You can mark tracked files (by aspect) as to be synced with the preferred state or not. * You can edit the preferred state for this point in the virtual tree. * Continuations are shown as: readme: has become readme.old (machine 2) moved to ../Archive/ (machine 2) perms -rw-r–r– (machine 2) CONFLICT! (preferred version) (n.b. all start unpreferred) perms -rw-rw-r– (machine 3) CONFLICT! (unpreferred version) readme: was readme.new (machine 2) moved from ../InPreparation/ (machine 2) - propagating aspect values are highlighted in green, to-be-overwritten values in red - moves show up under both filenames, can be changed either place, changes show up both places - there needs to be a way to do batch changes, should probably be designed after need is understood - easy to hop to other mentioned directories, easy to go back too

What if you make a new file A that is supposed to be synced with a new file B on other machines but you put A in the wrong place? How do you fix that? You will need to say that A has a preferred location somewhere else. That's all.

If there is a conflict you can say that one of the tracked files should be moved to a different name. Then you have both of the files, and each one inherits the old history, so you can merge either way if you want.

5.2 Graft

There are grafts of actual device directory trees onto the virtual tree. A graft is like a mounted filesystem in a unix machine's directory tree. But there it hides the previous contents, replacing it with the mounted contents, whereas in McSync, both contents are there, and McSync helps to keep them identical.

A stored file can be grafted to multiple points on the virtual tree, and multiple stored files can be grafted to the same point on the virtual tree. However, no chain of nested subdirectories, hopping back and forth between actual and virtual, may be cyclic. In other words, the result of an ideal sync must be finite.

This turns out to be very tricky to detect, see algorithm "How to detect cyclic grafts" below.

A graft point is where an actual file path is mapped to a virtual file path. A prune point belongs to a graft, and indicates a position within the subtree where the mapping ceases. Prune points may be patterns, such as "any dir containing a file named .svn".

5.3 Backup Devices

A device can be marked as a backup device. The idea is it would store every version of every file. Backup device needs to be able to store many copies of each file, each marked with source device and scan time when that version was noticed.

Multiple backup devices can be specified. On a backup device, no version would be erased (even if clearly unpreferable) unless it is known to be on at least n/m backup devices. Default is n=min(m,1), m=0. Nah, you never really know what's on another device. It would be tricky to avoid simultaneous erasures, each device assuming the other one has it. A graft (or pruned subtree) can be marked as keeping backups of everything. In other words, it wants all versions of everything from all other devices.

That would be strong backup. Weak backup would just be that it syncs like any other machine, but somehow keeps a record of its own older states, like if a backup system is observing that device. This would not allow recovery from an incorrect conflict resolution.

Maybe the mess is simply instructions, since instructions are file contents or metadata, plus the associated history. A backup device would just store a mess of instructions. To restore a previous version, you just execute that instruction. (The current history would treat any "reverting" change as a new local edit, but the historical instructions would remain in the backups, untouched.) Instructions already have a place to be stored until executed, so a backup system would simply store all the instructions it can get its hands on, but not execute most of them or delete any of them. It would not have to look different from any other graft.

Is a one-step undo possible in any sensible way? Well, an instruction to revert can be kept for each instruction executed. This is kind of like a weak backup system that only remembers one step back. This could be the default for every graft, kind of like emacs keeping one $\sim$ file by default. I don't think this is very useful. There could more generally be options about how much history to keep. But the first thing to implement is a complete backup (all versions) approach.

Part of keeping instructions is making sure you have any file version that the instructions assume are present. Instructions have a way of including such files anyway.

5.4 Propagation of Source and Configuration Files

settings are synced in the normal way, first

sources are synced in a different way (since user doesn't change them, and both ends need to match to run at all)

Chapter 6

Mobility Issues

6.1 separation of controller from large record keeping

6.1.1 Types of Processes

Commander The commander is the TUI or GUI or batch preferences or whatever is making the decisions. The program started by the user is the commander, running on the machine expected by the user, like any executable. The commander communicates with the headquarters, receiving just enough info for display, and providing just enough info for the headquarters to run the show. (Commander might be handheld device, while headquarters is big and powerful.) The commander is the only one who uses the virtual tree.

Headquarters The headquarters runs the history comparison algorithms, does history merging, creates instructions, and generally tells the workers what to do, based on what the commander tells it to do. Workers send info in to the headquarters (when requested), and the headquarters sends info back out to the workers. Conceivably the headquarters could tell workers to send data directly among themselves (possibly setting up further connections, so the network is not a tree), but we won't try that yet.

Workers A worker accesses files on one device, in response to requests from (and communicating results to) the headquarters. Two workers could handle two McSync locations on one machine.

Every device has a local router. The routers are the only ones who know whether messages actually need to be sent to other places, and the only ones who send them. Other processes just hand their messages to and from the local router. The routers are like the post office. Nobody else has to worry about how their message gets to its destination, they just say who it's for. The network topology for now is a tree.

6.2 Hopping from machine to machine

Check out "Using rsync through a firewall"

It has nothing to do with rsync and everything to do with making multiple login hops programmatically possible.

`http://samba.anu.edu.au/rsync/firewall.html`

6.3 Locations

By a location, we mean a machine in terms of how it can be accessed from the internet. If a laptop is taken from hotel to hotel, it is at a different location each time. If a USB stick is taken from one internet cafe to another, it is at a different location each time. If an external drive is mounted at different points at different times, its location changes accordingly.

Locations are important because they let us find devices. One location can even be connected to more than one device (multiple drives can be mounted somewhere). When McSync is running, it forms a network of processes at different locations all connected by a message routing system. There is a worker thread for every device, a router thread for every machine, an optional algorithm thread on each machine and an optional tui thread on each machine. The normal situation is that the tui and algorithm threads are on the machine where the user invoked McSync.

As it gets started, McSync knows of one device connected to the location it is running on (because the executable lives in the McSync directory for a device, and the executable can find this directory). It can attempt to figure out its machine or location, but what it really wants to do is to find other devices where McSync can be invoked.

Some devices live at fixed locations. Some locations are only visible from certain other locations. A device is typically either fixed (desktop, external drive) or mobile (laptop, USB stick). If a fixed device moves, the user can update how it should be reached from other locations, and how it reaches other locations, and this info propagates out. A mobile device has a list of familiar locations where it might be likely to be. These are just two modes of use, there is no distinction from McSync's point of view, except maybe that a fixed device is sure of its location while a mobile device has to try to figure it out, and I'm not sure how to figure it out.

Except during configuration setup (creating a new device), a location can only be reached if it has a device connected. Otherwise there is no McSync executable there to talk to. If the way to connect from A to B is via several ssh hops with no McSync on the intermediate machines, then from McSync's point of view, this is just a single connection between two locations, A and B.

So every device has a list of other devices that it is willing to try to reach (if they are not yet connected) via various locations. Some locations are worth trying even if you have only seen other devices (besides the one you're looking for) there before (e.g. USB mount point).

In fact, locations can be grouped into networks. For example, there are various locations in the internet network, locations in my home network, in my behind-firewall work network, one on my roaming laptop, and so on. A given location has access to certain networks (most can access internet, but roaming laptop might have access to no networks except itself, for example). For each network you can access, there are locations you can try to access. You always have to try a device locater (IP address)* (dir) and see what you get (also df might give you places to guess?). Since different USB sticks could be mounted at the same place (under the same name) at different times, you have to check what device you got, and make sure you're not already otherwise connected to it.

If there's a computer you want McSync to be aware of (take certain actions from, use as a branching point, etc.) even if you don't sync anything there, you'll need to have McSync running there, which means the executable is on a device, so that location is a device like any other. Nothing there needs to be grafted onto the virtual tree, so it is natural to sync nothing there. There is no branching except at devices, so all connections are immediate. Multi-hop connections occur through intermediate devices which have been reached, so every connection to another machine is (from McSync's point of view) a single-hop (though perhaps a multi-machine / multi-ssh) connection to another device.

This all seems to boil down to:

- Each device has a list of device locaters that it is willing to try. Each is for one or more devices.
- Device locaters can be grouped into 'networks'. Then each device has a list of networks it is willing to try.
- Networks can include other networks as well as device locaters.

```
   volume SP internet INI
// machine is named SP, willing to try to access devices on networks “internet”,
“INI”, and “SP”
   internet:login.ini.uzh.ch‘15‘$ :cook-sp‘30‘$ :~/Common/Coding/Unix/McSync/McSyncDA2
// machines on “internet” access SP like this
   INI:cook-sp‘$ :~/Common/Coding/Unix/McSync/McSyncDA2
// all machines on “INI” willing to try to access SP like this
   volume usb1 internet?
// device is named usb1, is willing to try to access devices accessible from
“internet” if user gives ok
   SP:/Volumes/usb1/.mcsync
// SP willing to try to access usb1 like this
   volume handheld (SP) internet
// handheld will use SP for its headquarters / algorithm
```

Boiling it again, we get:

- Each machine has a list of networks that it might be able to reach.
- Each of these has zero or more addresses (MLSs) where the machine might be found on that network.
- Each machine has a list of one or more file locations where it might find McSync.
- Each machine has a list of one or more file locations where there might be a mountable drive.
- There is a list of mobile drives that might be mounted at these points.
- Scripts can be provided to check whether you're on the machine or network you suspect, before taking actions that act on this suspicion.
- A network can have a default script, and a (machine,network) can override it.
- An address (MLS, McSync locator string) is either ssh:user@hostname: /.mcsync or expect:scriptname

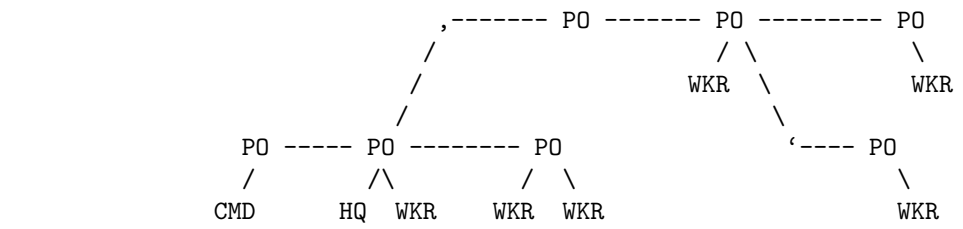
A device locater has 0 or more hops each with (a) an IP address, (b) a prompt to wait for, and (c) how many seconds to wait for it. If c is missing it is the default. If b is missing, it just waits for the full number of seconds. Finally, after all these hops, it has a path to McSync's directory on that device.

If McSync is started in "wait" mode, then instead of automatically trying to connect everywhere, it just gives a list of where one would like to connect from/to, and the user can ok any they like. If automatic, then the central machine tries to contact every machine it can reach directly. When one attempt times out, the next is tried for the same target. If all attempts (perhaps 0) have failed, it starts trying multi-hop routes, breadth-first. Each time a new attempt is to be made, it is recomputed based on the current set of connected devices. Whenever a device is reached, the central machine stops trying to reach the reached device (although it may continue trying to reach the device it was trying to reach, if that was different from the device it did reach).

Only CMD is trying to reach devices. Each device maintains a list of attempted routes. Given the specs and the current set of connected devices, it tries the first untried single-hop connection it can find, then the first two-hop connection it can find, and so on.

6.4 McSync's Internal Network

McSync starts up remote processes on other machines and maintains a communication network between these processes. The communication network takes the form of a tree, rooted at the initial process, which is where the CMD is. A different machine (perhaps one with better bandwidth) may be chosen to run HQ. CMD has responsibility for setting up the network.



In that picture, each PO represents a machine, and each WKR represents a device. A PO generally has a WKR. CMD represents the user interface or batch process controlling what happens.

HQ represents the main machine doing comparisons, but it will probably be phased out and moved to exist only in the CMD's view of things: CMD will simply tell machines to send records to each other, or to compare records. Eventually, multiple CMDs should even be able to coexist.

```
(up the tree) SR ---- plug ----.    ,---- plug ---- WKR 5 (hard disk)
(down to 3,4) SR ---- plug ---- PO 2 ---- plug ---- WKR 6 (usb)
  (down to 7) SR ---- plug ----'    '---- plug ---- maybe HQ and/or CMD
```

Note that both sides of a plug use the same plug struct. The two threads "own" different parts of the struct and its queues.

When McSync is invoked on a machine, main() transfers control to the router (PO), which sets up some plugs (each with a separate thread for the other side) and then starts routing among them. One of the plugs is for the machine worker (MW), which is like a worker for the PO, handling requests related to the PO. The PO thread itself just shuffles messages around from outboxes to inboxes. It doesn't have an address and cannot be talked to. If you could talk to it, requests could pile up, so we would want plug infrastructure. We get the plug infrastructure by letting MW be the one you talk to. PO and MW are like two rooms in the post office: PO is the mail sorting room, and MW is the correspondence and administration office.

A plug is a struct with various parts: some parts are used by the recruiter while setting up the remote connection before communication can start, some parts are the queues used by the communicating parties, some parts help the router tell whether this is the plug a message is looking for. Think of it as a pair of cubbyholes in the router room, made out of wood. This wooden object has installation and kid-making instructions written on the top, holes for transferring stuff, and an address (or many) for reference during use. The kid (on the other side of the cubbyhole wall) takes things out of the outbox and puts things in the inbox. Plugs for communicating with remote places have stream cables plugged in, which the kid uses.

One of the communicating parties is always the router, and the other is called the kid because the router spawns all the other threads it communicates with.

The master McSync PO spawns a CMD and a WKR. Later, it might set up a connection to a remote device, REMOTE (SR). Each slave PO spawns

a PARENT (SR) and a WKR. Any PO might spawn a RCTR, and if that succeeds then it converts into a REMOTE. Any PO might spawn a HQ, if no other PO has one. An HQ can be shut down if it has no ongoing activities to manage. CMD, HQ, RCTR, and WKR are local kids. PARENT and REMOTE use multiple threads to handle stream connections between machines. PARENT is run on a slave for communicating with the higher parts of the tree. REMOTE is for communicating down a particular branch of the tree.

List of agents:

- CMD - user interface
- HQ - headquarters (receives scans, acts as database for CMD, sends identification, gossip, and instructions)
- PO - post office (router)
- MW - machine worker (can report PO info, sets up new connections)
- RCTR - recruiter (attempts to reach remote machines, frequently freezes)
- SR - shipping/receiving (thin agents that just forward everything to/from some streams) (REMOTE, PARENT)
- DW - device worker (does scans, reads/writes histories, executes instructions)

6.5 Network and Remote Machines

Reasons to maintain a connection: Can display connectivity status. Only need password once. Reasons not to: Ok to connect for scan, disconnect for human thought, and connect for action. Solution: Allow each connection link to specify an optional timeout and/or heartbeat rate.

If we have different threads that both want to use a remote machine, it's easier if they don't have to coordinate, but each just contact the remote machine independently and use it simultaneously. This shouldn't really happen, unless there are two storage locations on that machine, and then the two threads are probably ok, as there are likely to be two separate disks. If a user is having trouble with thrashing, they should treat a single drive as a single storage device. Syncing stuff on two partitions of a drive might be awkward, but isn't it always? If it's a problem, there can be an option to indicate that two storage devices are sharing some kind of bandwidth and should be treated serially. Or, both partitions can be the same storage device. You can always have multiple locations on a single storage device be grafted onto the same point in the virtual tree.

So for a scan, invoke remote McSync and communicate with it on its stdin/stdout. Report progress to parent thread every so often. Finally, give parent pointer to history.

Put it under user control! Let them load a previous scan rather than doing a new scan, if they want. Let them connect without doing a scan if they want. Let them disconnect at any time.

Cpus/machines are only useful as networked nodes, for transferring/comparing data. Imagine all data as being on USB sticks. Imagine computers as being dhcp'd laptops, with 0 or more of the USB sticks currently plugged in.

Each storage device has a location (directory) for McSync to keep track of stuff. Various devices on various partitions (or whatever) of that device will be treated by a single executable. Of course, the machine may have to make the appropriate executable first. Not much point in propagating executables.

We have a storage device, located by its root, a directory. We find that directory by: (ssh to machine)*, look for directory Kind of like cook@dna.caltech.edu:somedir but can also be cook@to.dna.caltech.edu:cook@dna.caltech.edu:somedir In this case to.dna.caltech.edu would not run McSync or store any info.

So we need a way to create threads, that can do one of two things: 1. ssh to another machine, and then send all info there / get all info from there 2. look at files and do stuff with them

The problem is that USB sticks (as well as the laptops they are plugged into) are in different places at different times. And yet the USB stick has all the information about how to reach the others. So it needs to have a list of things to try. If each reached site then tries to go further, things will go in circles. So when a machine is reached and McSync is started as a slave, it should check for any already-running McSyncs under the same user, sending the pids back to the master, who keeps a collection of pids of all the slaves. If the pid matches, then some kind of file-writing test is done to see if it really is the same, and if it is, then the new one aborts itself and the old one is told to find whatever the new one was finding.

Of course it should be possible for two independent McSyncs to be running simultaneously on the same machine (even with the same working directory) as long as they don't have the same master. This is when lock files come into play. The device should only be scanned by one process at a time, since the scans need to be strictly ordered. And of course syncing or updating the history should only be done by one process at a time. But it should be fine to leave McSync running on each of several machines, with each instance connected to all the machines.

6.6 Processes and Communication

The master process starts a TUI thread, an algorithm thread, and a local router, which manages the communication between them.

A router together with its worker constitute the processes operating on a McSyncDeviceArchive directory, and together they form McSync's center of operations for the device. Remote processes start a router. A router at startup creates a local worker, and it can try to start remote routers. It publishes a list of locations that it is attempting to reach or has reached. It can take postcards and

packages. A postcard is stored locally before being forwarded asynchronously to its destinations, while packages are delivered through direct stream connections (blocking until all those connections are available, and connections that are part of another package distribution and are still just in the "connecting" stage can be booted by a higher-hash distribution).

A worker can use a lock file during operations such as updating its history information.

The master process communicates with the TUI thread mainly through the messaging system (although shared access to data structures would be silly to avoid (although the TUI might be on your PDA while the master is on your server, or the TUI might be a GUI while the master is a command-line utility, so maybe it should be avoided anyway)), so that it can easily be replaced by another kind of UI, preferably as a completely separate process, later. The TUI can first try to connect to machines and then as you see what it could connect to you can choose the master.

For each master or worker, there is a queue of commands and the status of the currently-being-processed command.

– Possible Protocals –

For info we want to push, like commands and results, we don't use pipes and streams, because we want to allow the data to pile up before being read. So we use data structures, and when the structure is ready we set a flag saying so.

For pullable info, like progress status, we avoid the need for locks by using a one-byte pointer to point to the latest info, and we assume a cycle of writes won't overtake a single read. If writes get ahead of reads then we simply skip writes (except last one).

Single-flag radios for inter-thread blocking communication: (blocking = writes must wait for previous read to finish) 0 = no message ready - receiver must wait - sender can create message and then set to 1 1 = message ready - sender must wait - receiver can read message and then set to 0 (It doesn't matter whether flag is a bit, a byte, or bigger, since only one bit will matter.)

Queues for inter-thread non-blocking communication: Head of queue is processed by receiver, tail is processed by sender. Flag NextExists can indicate whether the next pointer is valid yet. Flag being false is cleaner form of traditional next==NULL technique. After sender sets flag, it can't touch anything else in that struct (or in pointed-to struct, except next and flag), and receiver is free to free it. Receiver always holds on to queue head until flag indicates it can process the next one.

– Plugs –

Ok, that last one is the only one we use. We use the idea of a plug. A plug (typical var name) is a connection (type). It is simply a pair of message queues for two-way communication. A router has some plugs to various processes or other routers. Plugs are used by routers. Routers are connected in a tree. Routers have various plugs for processes they might want to communicate with. McSync runs either in master mode or in slave mode. Either way there is just one router. Although other threads view it as the post office, really it is in charge – it creates the other threads as needed, and anyway there is always exactly one

router, regardless of whatever else is needed locally. The router always starts a worker thread (connected through its worker plug). Only in slave mode does the router have a parent plug. The parent plug points back up the tree. The master mode router is the root of the tree and has no parent plug.

– Message types –

The master process starts the UI process, which can tell the master process to:

- connect to a storage device (telling it what history and scans we have)
- request a scan from a storage device at a given point with given prune pionts
- indicate a preferred aspect value at a point in the virtual tree

The master process can give the UI process:

- a requested scan

a storage device can send:

- an updated history

– Data formats used –

1. Gossip / History: (aspect name, aspect value, (machine, file tracking number, time, issame)*)* [with identical subtree optimization]

2. Scan: Like scan I/O, but uses a reference scan and where a subtree is identical, just says so instead of sending it.

3. Guidance: same as history

4. Instructions: this is like history too, because it gives a (new) aspect value along with its history

The data formats are all the same: A history. A scan just has no (m,f,t,i) quads,

– UI API –

```
------------------------------------------------------------
Steps in establishing a connection [$1:]$2:$3:$3:$4(dir)
L/I/R = local/intermediate/remote
T = TUI thread
A = algorithm thread
W = worker thread
P = plug thread
C = child process

    expect: script name OR ssh: machine, user
        user@machine:~/.mcsync
        @laptop2:user@machine:user@machine:~/.mcsync
            $1 = where to hop from -- not fully implemented
            $2 = first machine to ssh to
```

```
            $3 = continuing machines to ssh to
            $4 = final McSync directory
        sp2sp;running:min;$:AC;$:<input>
            not fully implemented yet

::TUI tells algo to connect to a certain device::
LT  TUIprocesschar  receives 'c'onnect command, sends "newplugplease" (NPP1/2)
                    message to algorithm w/ deviceid string(s)
(should be to some MW, w/ MLS (McSync locator string))
LA  algomain        receives NPP1 (devid) or NPP2(dstid,srcid) and calls reachfor

::algo asks worker to create a connection to the device::
LA  algo_reachfor   sends NPP (deviceid + routeraddr) to worker, uses
                 $1 deviceid to find machine and set its routeraddr
---                 at this point we go to whatever PO should try the connection,
                    could be local or remote.  we call it intermediate.
IW  workermain      receives NPP message and calls channel_launch
(should be I.MW)
IW  channel_launch  creates the new plug, sets target_machine to the machine with
                    the given deviceid, sets routeradder (on intermediate machine)
(should use MLS)
IP  thread_main     recognizes plug as needing connection and calls reachforremote

::dedicated thread tries to actually reach device::
IP  reachforremote  does regular scrolling, gives birth, gives further commands
                    to process, waits for McSync to show signs of life, sends
              $3,$4 router address
IP  givebirth       forks, sends child (one-way) to firststeps
IC  firststeps      fixes pipes
IC  transmogrify $2 become ssh or whatever through execl
---                 keep receiving commands from reachforremote
RC  main            prints messages to show signs of life, receives router address
RC  routermain      tells channel_launch to create a parent plug and a worker
                    plug with the given plug id
RC  channel_launch  creates the new plug, sets target_machine and routeraddr
RW  thread_main     sees it is the worker_plug
RW  workermain      sends algorithm a "workerisup" message
---                 every hop along the way adds the worker's plug number to the
                    thisway it came from

::communication is up::
LA                  algorithm receives this and asks worker for deviceid string
RW                  worker sends deviceid
LA                  algo finds machine with given deviceid, sets status_connected
LT                  TUI shows machine is connected
-------------------------------------------------------------
```

Chapter 7

Source Code Organization

McSync is modular to a certain extent. For example, it can grow its own ad-hoc network to connect reachable devices, but this networking code is completely disjoint from the algorithmic code that uses the network. This chapter explains how the source code is organized, and how the different parts of the code interact with each other.

7.1 Preferences

The preferences of McSync do not only store options for customizing the user interaction to suit the user's personal taste. They also store information about where to look for different devices, and how those devices map onto the virtual tree.

McSync is faced with users invoking it at different locations at different times. It tries to keep a unified set of stored preferences, but of course keeping these preferences synced is no easier than keeping anything else synced. Even worse, it is the preferences themselves that specify how things should be kept in sync, so outdated preferences might themselves prevent the preferences from being updated correctly.

McSync simply does the best it can do: Based on the preferences available where it is invoked, it tries to reach other machines and devices. Every device contains a graft of McSync internal data, which includes the preferences. This graft is scanned automatically whenever a new device is reached, and the user is alerted if anything is not up to date. If the user tries to update to new preferences which would not have found a connected device, a warning is provided, but the device is not disconnected.

McSync always uses the current preferences of the device on which it was invoked.

7.2 Networking

The networking layer tries to establish connections to other machines. It creates a network where each node in the network is a machine running McSync, responsible for zero or more devices. Each connection in the network is a two-way communication link. For now, the network structure is always a tree.

For McSync to be run, there needs to be an executable, which is naturally in a McSync directory on a storage device. So we might expect that every executable has its own private device to work with. But it is possible that this is a storage device that is available to many machines, and the machines might nonetheless have different network connectivity properties, so that McSync might need to use two of these machines in order to build its network. Thus it is conceivable that a connection to a new machine does not actually provide any new device connections.

The networking layer can route device requests such as a request to do a scan, merge some gossip, or follow some instructions. The networking layer itself does not care about the contents of these requests, but simply views them as packets to deliver to the machine responsible for that device. Requests to send information, such as scan results or file contents, are also viewed simply as data streams from one device to another.

The network can send packets of arbitrary size from any machine or device to any other. Every device and every machine has an address (a small integer) which is used for routing.

7.3 Device Access

The device access layer can read and write file metadata as well as the file contents. It reads when taking a snapshot, and writes when carrying out instructions.

7.4 Data Transfer

- knows McSync formats.
- transfers various types of files between machines.
- responsible for any compression or delta optimizations.

7.5 User Actions Layer

- knows about virtual tree.
- can request snapshots.
- can generate guidance.

- can request syncs.
- can request new connections.

7.6 Algorithm Layer

- knows McSync formats.
- can incorporate scans into histories.
- can compare histories.
- can generate instructions.

Index

ancestor, 27
arbitrary distinguishability theorem, 72
aspect, 22

backups, 77

comparing histories, 32
computer, 21
consistency theorem, 66
CPU, 21
ctime, 73

disk drive, 21
distinguishability, 72

extended attributes, 23

file, 22, 27
file contents, 23
file parts, 22
filename, 23
flash drive, 21
free preferability theorem, 72
full history, 30

ghost deletion theorem, 71
global improvement theorem, 66
gossip, 33
graft, 77
guidance, 35

hard drive, 21
history, 30
history comparison, 32

identification, 35
inode, 73
inode and ctime theorem, 73
instructions, 39

key-value pairs, 25

local improvement theorem, 66
lossless theorem, 64

machine, 21
merge tools, 25
missing, 22
modification time, 23

naive merge theorem, 66
new version, 29
not visible, 22
noticing, 38

old version, 29
optimality theorem, 65

parent directory, 23
permissions, 23
preferability, 29, 72

raw file, 27

scan, 29
sets, 26
snapshot, 29
storage device, 21
sync, 39

tracked file, 27
tracking number, 28

version, 29
virtual file, 27
virtual tree, 76

Bibliography

[1] *Collected Sayings of Me*, M. Cook, My Journal 1:7-9 (January 1, 2000)
This article shows I'm right.