When File Synchronization Meets Number Theory

Antoine Amarilli, Fabrice Ben Hamouda, Florian Bourse,
David Naccache, and Pablo Rauzy

École normale supérieure, Département d'informatique
45, rue d'Ulm, F-75230, Paris Cedex 05, France.
`surname.name@ens.fr` (except for `fabrice.ben.hamouda@ens.fr`)

Abstract. In this work we [to be completed by David]

1 Introduction

This work revisits *set reconciliation*, a problem consisting in reconciling two multisets whiles minimizing communication complexity. Set reconciliation has many practical applications the most typical of which is the efficient incremental backup of hard drives..

Set reconciliation admits already efficient and elegant solutions. For instance, [1] presents a reconciliation protocol whose expected computational and communication complexities are linear in the number of differences in the reconciling hosts. Note that computation must also be at least be linear in the size of the reconciled multisets.

We refer the reader to [1,2,4] (to quote only a few references) for more on the problem's history and existing solutions.

To efficiently synchronize directories, we propose a new protocol based on number theory. In terms of asymptotic complexity, the proposed procedure is comparable to prior publications [1] (that anyhow reached optimality) but its interest lies in its simplicity, novelty and the possibility that specific implementations would offer *constant*-factor gains over alternative asymptotically-equivalent solutions.

Interestingly, using the approach described in [3] the correctness of the protocols proposed in this paper is reducible to the cryptographic security of specific signature padding schemes.

2 The Content Synchronization Protocol

2.1 A Few Notations

We model the directory synchronization problem as follows: Oscar possesses an old version of a directory $\mathfrak{D}$ that he wishes to update. Neil has the up-to-date version $\mathfrak{D}'$. The challenge faced by Oscar and Neil[1] is that of *exchanging as little data as possible* during the synchronization process. In reality $\mathfrak{D}$ and $\mathfrak{D}'$ usually differ both in their files and in their tree structure.

In tackling this problem this paper separates the *"what"* from the *"where"*: namely, we disregard the relative position of files in subdirectories and model directories as multisets of files. Let $\mathfrak{F}$ and $\mathfrak{F}'$ denote the multisets of files contained in $\mathfrak{D}$ and $\mathfrak{D}'$. We denote $\mathfrak{F} = \{F_0, \dots, F_n\}$ and $\mathfrak{F}' = \{F'_0, \dots, F'_{n'}\}$.

[1] Oscar and Neil will respectively stand for *o̲ld* and *n̲ew*.

Let Hash denote a collision-resistant hash function[2] and let F be a file. Let $\texttt{NextPrime}(F)$ be the prime immediately larger than $\texttt{Hash}(F)$ and let u denote the size of NextPrime's output in bits. Define the shorthand notations: $h_i = \texttt{NextPrime}(F_i)$ and $h'_i = \texttt{NextPrime}(F'_i)$.

TODO(amarilli): use the uniform nextprime (discussion of relative costs with respect to (1.) hashing costs and (2.) finding the next prime costs)

2.2 Description of the Basic Exchanges

Let t be the number of discrepancies between $\mathfrak{F}$ and $\mathfrak{F}'$ that we wish to spot, *i.e.*:

$$t = \#\mathfrak{F} + \#\mathfrak{F}' - 2\#(\mathfrak{F} \bigcap \mathfrak{F}')$$

We generate a prime p such that:

$$2^{2ut+1} \le p < 2^{2ut+2} \tag{1}$$

Given $\mathfrak{F}$, Neil generates and sends to Oscar the redundancy:

$$c = \prod_{i=1}^{n} h_i \bmod p$$

Oscar computes:

$$c' = \prod_{i=1}^{n} h'_i \bmod p \quad \text{and} \quad s = \frac{c'}{c} \bmod p$$

Using [7] the integer s can be written as:

$$s = \frac{a}{b} \bmod p \quad \text{where the } G_i \text{ denote files and } \begin{cases} a = \displaystyle\prod_{G_i \in \mathfrak{F}' \wedge G_i \notin \mathfrak{F}} \texttt{NextPrime}(G_i) \\ b = \displaystyle\prod_{G_i \notin \mathfrak{F}' \wedge G_i \in \mathfrak{F}} \texttt{NextPrime}(G_i) \end{cases}$$

Note that since $\mathfrak{F}$ and $\mathfrak{F}'$ differ by at most t elements, a and b are strictly lesser than 2^{ut}. Theorem 1 (see [2]) guarantees that given s one can recover a and b efficiently (this problem is known as *Rational Number Reconstruction* [4,8]). This is done using Gauss' algorithm for finding the shortest vector in a bi-dimensional lattice [7].

Theorem 1. *Let $a, b \in \mathbb{Z}$ such that $-A \le a \le A$ and $0 < b \le B$. Let $p > 2AB$ be a prime and $s = ab^{-1} \bmod p$. Then given A, B, s, p, one can recover a and b in polynomial time.*

Taking $A = B = 2^{ut} - 1$, (1) implies that $2AB < p$. Moreover, $0 \le a \le A$ and $0 < b \le B$. Thus Oscar can recover a and b from s in polynomial time. By testing the divisibility of a and b by the h_i and the h'_i, Neil and Oscar can precisely and deterministically identify the discrepancies between $\mathfrak{F}$ and $\mathfrak{F}'$ and settle them.

Formally, this is done as follows:

[2] *e.g.* SHA-1

Oscar		**Neil**
	$\xrightarrow{c}$	
		computes a, b if a doesn't factor into a product of h_i's then output $\perp$ and halt else let $\mathfrak{S} = \{F_i' \text{ s.t. } a \bmod h_i' = 0\}$
	$\xleftarrow{\mathfrak{S},b}$	
deletes files s.t. $b \bmod h_i = 0$ adds $\mathfrak{S}$ to the disk		

As we have just seen, the "output $\perp$ and halt" should actually never occur as long as bounds on parameter sizes are respected. However, a file synchronization procedure that works *only* for a limited number of differences is not very useful in practice. In the next subsection we will explain how to extend the protocol even when differences exceed t, the informational capacity of the modulus p used.

2.3 The Case of Insufficient Information

To extend the protocol to an arbitrary number of differences, Oscar and Neil agree on an infinite set of primes $p_1, p_2, \ldots$ As long as the protocol fails, Neil will keep accumulating information about the difference as shown in appendix A. Note that no information is lost and that stockpiled information adds up until it reaches a threshold that suffices to identify the difference.

To determine after each p_i round if the synchronization is over, as the interaction starts Neil will send to Oscar $\texttt{Hash}(\mathfrak{F}')$. As long as Oscar's state does not match this target hash $\texttt{Hash}(\mathfrak{F}')$, Oscar will continue the interaction.

3 Efficiency Considerations

In this section we explore two strategies to reduce the size of p and hence improve transmission by *constant factors* (note that from an asymptotic complexity standpoint, nothing can be done as the protocol already transmits information proportional in size to the difference to settle).

3.1 Probabilistic Decoding: Reducing p

Generate a prime p smaller than previously, namely:

$$2^{ut+w-1} < p \leq 2^{ut+w} \tag{2}$$

where $w \geq 1$ is some small integer (say $w = 50$). For large $\eta = \max(n, n')$ and t, the size of the new prime p used in this section will be approximately half the size of the prime p generated in section 2.2. The new redundancy c is calculated as previously and is hence approximately twice smaller. As previously, we have:

$$s = \frac{a}{b} \bmod p \quad \text{and} \quad \begin{cases} a = \displaystyle\prod_{G_i \in \mathfrak{F}' \wedge G_i \notin \mathfrak{F}} \texttt{NextPrime}(G_i) \\ b = \displaystyle\prod_{G_i \notin \mathfrak{F}' \wedge G_i \in \mathfrak{F}} \texttt{NextPrime}(G_i) \end{cases}$$

and since there are at most t differences, we must have:

$$ab \leq 2^{ut} \tag{3}$$

The difference with respect to the protocol of section 2.2 is that we do not have a fixed bound for a and b anymore; equation (3) only provides a bound for the *product ab*. Therefore, we define a finite sequence of couples of integers:

$$(A_i, B_i) = (2^{wi}, \lfloor \frac{p-1}{2A_i} \rfloor) \quad \text{where} \quad B_i > 1$$

For all $i > 0$ we have $2A_iB_i < p$. Moreover, from equations (2) and (3) there must exist at least one index i such that $0 \leq a \leq A_i$ and $0 < b \leq B_i$. Then using Theorem 1, given (A_i, B_i, p, s) one can recover a and b, and eventually determine the difference between $\mathfrak{F}$ and $\mathfrak{F}'$.

The problem is that (by opposition to the basic protocol) we have no guarantee that such an (a, b) is unique. Namely we could (in theory) stumble by sheer bad luck upon another (a', b') satisfying (3) for some index $i' \neq i$. We expect this to happen with negligible probability for large enough w, but this makes the modified protocol heuristic only.

To make the heuristic synchronization deterministic, the parties can can use the $\texttt{Hash}(\mathfrak{F}')$ protocol preamble mentioned in section 2.3.

3.2 The File Laundry: Reducing u

What happens if we shorten u in the basic protocol?

As illustrated by the birthday paradox, we should start seeing collisions. Let us analyze the statistics governing their appearance.

Regard `Hash` as a random function from $\{0,1\}^*$ to $\{0, \ldots, 2^u - 1\}$. Let X_i^1 be the random variable equal to 1 when the file F_i collides with another file, and equal to 0 otherwise. Clearly, we have $\Pr[X_i = 1] \leq \frac{\eta-1}{2^u}$. The average number of files which collide is hence:

$$\mathbb{E}\left[\sum_{i=0}^{\eta-1} X_i\right] \leq \sum_{i=0}^{\eta-1} \frac{\eta-1}{2^u} = \frac{\eta(\eta-1)}{2^u}.$$

For instance, for $\eta = 10^6$ files and 32-bit digests, the expected number of colliding files is less than 233.

That being said, note that a collision can only yield a *false positive* and never a *false negative*. In other words, while a collision may obliviate (make the parties blind to) a difference[3] a collision can never create an nonexistent difference *ex nihilo*.

[3] *e.g.* result in confusing `index.htm` and `iexplore.exe`.

Hence, it suffices to replace $\texttt{Hash}(F)$ by a chopped $\texttt{MAC}_k(F) \bmod 2^u$ to quickly filter-out file differences by repeating the protocol for $k = 1, 2, \ldots$ At each round the parties will detect new files and new deletions, fix these and "launder" again the remaining files.

Indeed, the probability that a stubborn file persists colliding decreases exponentially with the number of iterations k, if the MACs are random and independent between iterations. Assume that η remains invariant between iterations. Let X_i^ℓ be the random variable equal to 1 if the i-th file collides with another file during iteration ℓ, and equal to 0 otherwise. Let Y_i be the random variable equal to 1 when the i-th file collides with another file for all the k iterations, and equal to 0 otherwise, *i.e.* $Y_i = \prod_{l=1}^k X_i^\ell$.

By independence, we have:

$$\Pr\left[\,Y_i = 1\,\right] = \Pr\left[\,X_i^1 = 1\,\right] \ldots \Pr\left[\,X_i^k = 1\,\right] \le \left(\frac{\eta-1}{2^u}\right)^k .$$

Therefore the average number of colliding files is:

$$\mathbb{E}\left[\sum_{i=0}^{\eta-1} Y_i\right] \le \sum_{i=0}^{\eta-1}\left(\frac{\eta-1}{2^u}\right)^k = \eta\left(\frac{\eta-1}{2^u}\right)^k .$$

Hence the probability that least one false positive will survive k rounds is:

$$\epsilon_k \le \eta\left(\frac{\eta-1}{2^u}\right)^k$$

For the previously considered instance[4] we get $\epsilon_2 \le 5.43\%$ and $\epsilon_3 \le 2\cdot 10^{-3}\%$.

A more refined (but somewhat technical) analysis As mentioned previously, the parties can remove the files detected as different during the first possible iteration and work only with common and colliding files. Now, the only collisions which can fool round k, are the collisions of a files i and j such that files i and j have both already collided during all the previous iterations. We call such collisions "stutters". Let Z_i^ℓ be the random variable equal to 1 when file i has stutters during the ℓ first iterations.

Suppose that $\eta > 1$. Set $Z_i^0 = 1$ and write $p_\ell = \Pr\left[\,Z_i^{\ell-1} = 1 \text{ and } Z_j^{\ell-1} = 1\,\right]$ for all ℓ and $i \neq j$. For $k \ge 1$, we have

$$\begin{aligned}\Pr\left[\,Z_i^k = 1\,\right] &= \Pr\left[\,\exists j \neq i,\ X_{i,j}^k = 1,\ Z_i^{k-1} = 1 \text{ and } Z_j^{\ell-1} = 1\,\right] \\ &\le \sum_{j=0,j\neq i}^{\eta-1} \Pr\left[\,X_{i,j}^{k-1} = 1\,\right]\Pr\left[\,Z_i^{k-1} = 1 \text{ and } Z_j^{k-1} = 1\,\right] \\ &\le \frac{\eta-1}{2^u} p_{k-1}\end{aligned}$$

[4] $\eta = 10^6$,$u = 32$.

Furthermore $p_0 = 1$ and

$$\begin{aligned}
p_\ell &= \Pr\left[X_0^\ell = X_1^\ell,\ Z_0^\ell = 1 \text{ and } Z_1^\ell = 1\right] + \Pr\left[X_0^\ell \neq X_1^\ell,\ Z_0^\ell = 1 \text{ and } Z_1^\ell = 1\right] \\
&\le \Pr\left[X_0^\ell = X_1^\ell,\ Z_0^{\ell-1} = 1 \text{ and } Z_1^{\ell-1} = 1\right] \\
&\quad + \sum_{i \ge 2, j \ge 2} \Pr\left[X_{0,i}^\ell = 1,\ X_{1,j}^\ell = 1,\ Z_0^{\ell-1} = 1 \text{ and } Z_1^{\ell-1} = 1\right] \\
&= \Pr\left[X_0^\ell = X_1^l\right] \Pr\left[Z_0^{\ell-1} = 1 \text{ and } Z_1^{\ell-1} = 1\right] \\
&\quad + \sum_{i \ge 2, j \ge 2} \Pr\left[X_{0,i}^\ell = 1\right] \Pr\left[X_{1,j}^\ell = 1\right] \Pr\left[Z_0^{\ell-1} = 1 \text{ and } Z_1^{\ell-1} = 1\right] \\
&\le \frac{1}{2^u} p_{\ell-1} + \frac{(\eta-2)^2}{2^{2u}} p_{\ell-1}
\end{aligned}$$

hence:

$$p_\ell \le \left(\frac{1}{2^u} + \frac{(\eta-2)^2}{2^{2u}}\right)^\ell,$$

and

$$\Pr\left[Z_i^\ell = 1\right] \le \left(\frac{1}{2^u} + \frac{(\eta-2)^2}{2^{2u}}\right)^{k-1}$$

And finally, the survival probability of at least one false positive after k rounds:

$$\epsilon'_k \le \frac{\eta(\eta-1)}{2^u}\left(\frac{1}{2^u} + \frac{(\eta-2)^2}{2^{2u}}\right)^{k-1}$$

For the $(\eta = 10^6, u = 32, k = 2)$ instance considered previously we get $\epsilon'_2 \le 0.013\%$.

As one can see for a fixed k, ϵ'_k is a decreasing function of u. For a fixed u, ϵ'_k is also a decreasing function of k. Transmission, however, grows with u (bigger digests) and with k (more iterations). Hence, to optimize transmission, a careful determination of the

4 Theoretical Complexity and Algorithmic Improvements

In this section, we analyse the theoretical costs of our algorithms and propose some algorithmic improvements.

TODO(amarilli): we should compare the time complexity to that of the other paper, and, if we are better, insist on it. If we can combine this to "Practical set reconciliation", we should.

4.1 Theoretical complexity

Let $M(k)$ be the time required to multiply two numbers of k bits. We suppose $M(k+k') \ge M(k) + M(k')$, for any k, k'. We know that the division and the modular reduction of two numbers of k bits modulo a number of k bits costs $\tilde{O}(M(k))$ [1]. Furthermore, using naive algorithms, $M(k) = O(k^2)$, but using fast algorithms such as FFT [5], $M(k) = \tilde{O}(k)$. We note that the FFT multiplication is faster than the other methods (naive or Karatsuba) for number of about $10^4 \cdot 64$ bits (from gmp sources – if you find any better sources, it would be interesting...). And using such big numbers, the division and the modulo reduction algorithms used in gmp are also the ones with complexity $\tilde{O}(M(k))$.

Since p has $2ut$ bits, here are the costs:

1. (Neil) computation of the redundancy $c = \prod_{i=1}^{n} h_i \bmod p$, cost: $O(nM(ut))$, $\tilde{O}(nut)$ with FFT
2. (Oscar) computation of the redundancy $c' = \prod_{i=1}^{n} h_i \bmod p$, cost: $O(nM(ut))$, $\tilde{O}(nut)$ with FFT
3. (Oscar) computation of $s = c'/c \bmod p$, cost: $M(ut)$, $\tilde{O}(ut)$ with FFT
4. (Oscar) computation of the two ut-bits number a and b, such that $s = a/b \bmod p$, cost: $\tilde{O}(M(ut))$, using a new technique of Wang and Pan in [4] and [8]; however using naive extended gcd, it costs $\tilde{O}((ut)^2)$. @fbenhamo TODO However I do not know any software where it is implemented, nor the actual speed in practice, neither if this can be adapted for the polynomial case (this can be an advantage over the polynomial method for set reconciliation – but I think this is not the case, unfortunately, I have not access to interesting articles about polynomial rational reconstruction - but see p.139 of http://algo.inria.fr/chyzak/mpri/poly-20120112.pdf).
5. (Oscar) factorization of a, i.e., n modulo reductions of a by a h_i, cost: $\tilde{O}(nM(ut))$, $\tilde{O}(nut)$ with FFT
6. (Oscar) factorization of b, i.e., n modulo reductions of b by a h_i, cost: $\tilde{O}(nM(ut))$. $\tilde{O}(nut)$ with FFT

4.2 Improvements

It is possible to improve the complexity of the computation of the redundancy and the factorization to $\tilde{O}(n/tM(ut)$, $\tilde{O}(nu)$ with FFT [5]. To simplify the explanations, let us suppose t is a power of 2: $t = 2^\tau$, and t divides n.

The idea is the following: we group h_i by group of t elements and we compute the product of each of these groups (without modulo)

$$H_j = \prod_{i=jt}^{jt+t-1} h_i.$$

Each of these products can be computed in $\tilde{O}(M(ut))$ using a standard method of product tree, depicted in Algorithm 1 (for $j = 0$) and in Figure 1. And all these n/t products can be computed in $\tilde{O}(n/tM(ut))$. Then, one can compute c by multiplying these products H_j together, modulo p, which costs $\tilde{O}(n/tM(ut))$.

The same technique applies for the factorization, but this time, we have to be a little more careful. After computing the tree product, we can compute the residues of a (or b) modulo H_j, then we can compute the residues of these new elements modulo the two children of H_j in the product tree ($\prod_{i=jt}^{jt+t/2-1} h_i$ and $\prod_{i=jt}^{jt+t/2-1} h_i$), and then compute the residues of these two new values modulo the children of the previous children, and so on. Intuitively, we go down the product tree doing modulo reduction. At the end (i.e., at the leaves), we obtain the residues of a modulo each of the h_i. This algorithm is depicted in Algorithm 2 and in Figure 2 (for $j = 1$). The complexity of the algorithm is $\tilde{O}(M(ut))$, for each j. So the total complexity is $\tilde{O}(n/t\tilde{O}(M(ut))$.

Algorithm 1 Product tree algorithm

Require: a table h such that $h[i] = h_i$

Ensure: $\pi = \pi_1 = \prod_0^{t-1} h_i$, and $\pi[i] = \pi_i$ for $i \in \{1, \ldots, 2t-1\}$ as in Figure 1

```
$\pi \leftarrow$ array of size $t$
function PRODTREE($i$,start,end)
    if start $=$ end then
        return 1
    else if start $+ 1 =$ end then
        return $h$[start]
    else
        mid $\leftarrow \lfloor(\text{start} + \text{end})/2\rfloor$
        $\pi[i] \leftarrow$PRODTREE($2 \times i$,start,mid)
        $\pi[i+1] \leftarrow$PRODTREE($2 \times i + 1$,start,mid)
        return $\times$ PRODTREE(mid,end)
$\pi[1] \leftarrow$PRODTREE($1, 0, t$)
```

Algorithm 2 Division using product tree

Require: a an integer, π the product tree from Algorithm 1

Ensure: $A_i = A[i] = a \bmod \pi_i$ for $i \in \{1, \ldots, 2t-1\}$, computed as in Figure 2

```
$A \leftarrow$ array of size $t$
function MODTREE($i$)
    if $i < 2t$ then
        $A[i] \leftarrow A[\lfloor i/2 \rfloor] \bmod \pi[i]$
        MODTREE($2 \times i$)
        MODTREE($2 \times i + 1$)
$A[1] \leftarrow a \bmod \pi[1]$
MODTREE(2)
MODTREE(3)
```

5 Optimizing Parameters

The proposed process lends itself to a final fine-tuning. We list here some of the proposed research directions that could be investigated to that end:

5.1 Using a Smooth p

Comme explique dans un ancien email, je pense que l'on devrait utiliser un produit de petits nombres premiers au lieu d'un grand nombre premier p. Des l'instant que ces petits nombres premiers sont plus grands que les hashes, cela fonctionne. L'interêt est que l'on peut travailler modulo ces "petits nombres premiers" avec le CRT. Et en plus, la generation de ce modulo p (pas premier) est beaucoup plus rapide.

- The fact that there are not 2^u but $\frac{2^u}{u}$ primes.
- Faster hashing into the primes using $h - h \bmod \pi + k\pi$ where π is a product of small primes
- The fact that in the probability formulae $t_1 + t_2$ can be used instead of η
- The fact that the parameter u can be optimized in an adaptive way. As we go in rounds (i.e. generating a sequence $u_1, u_2, ...$)

6 Implementation

To illustrate the concept, the authors has coded and evaluated the proof of concept described in this section.

The executable and source codes of the program, called btrsync, can be downloaded from: `https://github.com/RobinMorisset/Btrsync`.

The synchronisation is unidirectional (clearer). The program consist in two subprograms: a bash script and a python script:

6.1 The Bash Script

A bash script runs a python script (describe below) on the two computers to be synchronized. If the computer is not the one running the bash script, the python script is executed through ssh. The bash scripts also creates two pipes: one from Neil stdin to Oscar stdout and one from Oscar stdin to Neil stdout. Data exchanged during the protocol transits *via* these two pipes.

6.2 The Python Script

The python script uses gmp which implements all the number theory operations required by Oscar and Neil, and does the actual synchronization. This script works in two phases:

Finding Different Files

1. Compute the hashes of all files concatenated with thier paths, type (folder/file), and permissions (not supported yet).
2. Implement the protocol proposed in Section **??** [add here a reference to the appropriate section in the paper] with input data comeing from stdin and output data going to stdout.

More precisely:

- Oscar sends it product of hashes modulo a first prime number p_1.
- Neil receives the product, divides by its own product of hashes, reconstructs the fraction modulo p_1 [can we elaborate more on what happens here? which functions in GMP are used to do the reconstruction?] and checks if he can factor the denominator using his hashes base. If he can, he stops and sends the numerator and the list of tuples (path, type, hash of content of the file) corresponding to the denominator's factors. Otherwise he sends "None" [is this the ASCII string "None"? if not what does he send precisely?].
- If Neil sent "None", Oscar computes the product of hashes modulo another prime p_2, sends it... CRT mechanism... [can we elaborate more on what happens here? which functions in GMP are used to do the CRT?]
- If Neil sent the numerator and a list of tuples, then Oscar factors the numerator over his own hash values. Now each party (Neil, Oscar) knows precisely the list of files (path + type + hash of content) that differs from the over party.

[please structure the following:]

2. synchronize all the stuff [this is not an expression we can use in a paper...]. This part is not completely optimized.

We just remove all folders Oscar should not have and create new folders.

Then we remove all files Oscar should not have and synchronize using rsync the last files.

We could check for move (since we have the list of hash of contents of files) and do moves locally.

We can even try to detect moves of complete subtrees...

Capture the following:

6.3 Move Resolution Algorithm

To reproduce the structure of Oscar on Neil, we have a list of file moves to apply. Sadly, it is not straightforward to apply the moves, because, if we take a file to move, its destination might be blocked, either because a file already exists (we want to move a to b, b already exists), or because a folder cannot be created (we want to move a to b/c, b already exists but is a file and not a folder). Note that for a move operation $a \to b$, there is at most one file blocking the location b: we will call it the *blocker*.

If the blocker is not present on Oscar, then we can just delete. However, if it exists, then we might need to move it somewhere else before we solve the move we are interested in. This move itself might have a blocker, and so on. It seems that we just need to continue until we reach a move which has no blocker or where the blocker can be deleted, but we can get caught in a cycle: if we must move a to b, b to c and c to a, then we will not be able to perform anything without using a temporary location.

How can we perform the moves? A simple way would be to move each file to a unique temporary location and then rearrange files to our liking: however, this performs many unnecessary moves and will result in problems if the program is interrupted. We can do something more clever so by performing a decomposition in strongly connected components of the *move graph* (with one vertex per file and one edge per move operation going from to the file to its blocker or to its destination if no blocker exists). The computation of the SCC decomposition is simplified by the observation that because two files being moved to the same destination must be equal, we can only keep one arbitrary in-edge per node, and look at the graph pruned in this fashion: its nodes have in-degree at most one, so the strongly connected components are either single nodes or cycles. Once the SCC decomposition is known, the moves can be applied by applying each SCC in a bottom-up fashion, an SCC's moved being solved either trivially (for single files) or using one intermediate location (for cycles).

The detailed algorithm is implemented as two mutually recursive functions and presented as Algorithm 3.

Algorithm 3 Perform moves

```
Require: $D$ is a dictionary where $D[f]$ denotes the intended destinations of $f$

$M \leftarrow []$
$T \leftarrow []$
for $f$ in $D$'s keys do
    $M[f] \leftarrow$ not_done
function UNBLOCK_COPY($f, t$)
    if $t$ is blocked by some $b$ then
        if $b$ is not in $D$'s keys then
            unlink($b$)                                   ▷ We don't need $b$
        else
            RESOLVE($b$)                                  ▷ Take care of $b$ and make it go away
    if $T[f]$ was set then
        $f \leftarrow T[f]$
    copy($f$, $d$)
function RESOLVE($f$)
    if $M[f]$ = done then
        return                                            ▷ Already managed by another in-edge
    if $M[f]$ = doing then
        $T[f] \leftarrow$ mktemp()
        move($f$, $T[f]$)
        $M[f] \leftarrow$ done
        return                                            ▷ We found a loop, moved $f$ out of the way
    $M[f] \leftarrow$ doing
    for $d \in D[f]$ do
        if $d \neq f$ then
            unblock_copy($f$, $d$)                        ▷ Perform all the moves
    if $f \notin D[f]$ and $T[f]$ was not set then
        unlink($f$)
    if $T[f]$ was set then
        unlink($T[f]$)
for $f$ in $D$'s keys do
    RESOLVE($f$)
```

An optimization implemented by `btrsync` over the algorithm described here is to move files instead of copying them and then removing the original file, because moves are faster than copies on most filesystems (because they don't need to copy the file contents).

6.4 Experimental Comparison to rsync

To demonstrate the benefits of our approach, we compared our btrsync implementation to the standard rsync on the following datasets:

synthetic A directory containing 1000 very small files containing the numbers from 1 to 1000.

synthetic_shuffled The result of applying a few operations to synthetic: 10 files were deleted, 10 files were renamed, and the contents of 10 files was changed.

source A snapshot of the btrsync source tree.

source_moved The result of renaming a big folder (several hundred of kilobytes) in source.

firefox-13.0 The source archive of Mozilla Firefox 13.0.

firefox-13.0.1 The source archive of Mozilla Firefox 13.0.1.

empty An empty folder.

We performed the measurements with rsync version 3.0.9 (used both as the standalone rsync and for the underlying call to rsync in btrsync). The standalone rsync was given the --delete flag to delete existing files on Oscar which do not exist on Neil. We passed the -I flag to ensure that rsync did not cheat by looking at the file modification times (which btrsync does not do), and --chmod="a=rx,u+w" as an attempt to disable the transfer of file permissions (which btrsync does not transfer). Although these settings ensure that rsync does not need to transfer the permissions, verbose logging suggests that it transfers them anyway, so rsync must lose a few bytes per file as compared to btrsync for this reason.

Bandwidth accounting was performed with the -v flag for rsync invocations (which report the number of sent and received bytes), and by counting the amount of data transmitted for btrsync's own negociations. The experiment was performed between two remote hosts on a high-speed link. The time measurements account both for CPU time and for transfer time and are just given as a general indication.

The results are given in table 3. The timing results show that btrsync is slower than rsync, especially when the number of files is high (i.e., the synthetic datasets). The bandwidth results, however, are more satisfactory. It is true that the trivial datasets where either Oscar or Neil have no data, rsync outperforms btrsync: this is especially clear in the case where Neil has no data: rsync immediately notices that there is nothing to transfer, whereas btrsync transfers information to determine the symmetric difference. On the non-trivial datasets, however, btrsync outperforms rsync. This is the case on the synthetic datasets, where btrsync does not have to transfer information about all the files which were not modified, and even more so on the case where there are no modifications at all. On the Firefox source code dataset, btrsync saves a very small amount of bandwidth, presumably because of unmodified files. For the btrsync source code dataset, we notice that btrsync, unlike rsync, was able to detect the move and to avoid retransferring the moved folder.

7 Haskell Implementation

(Antoine: je range ça dans sa propre section, mais j'imagine qu'on va le couper ou juste dire que ça existe.)

7.1 Program Structure

A proof-of-concept called Btrsync has been implemented in Haskell and is available at https://github.com/RobinMo

It is intended to work as a drop-in replacement of rsync for directories, taking as arguments two (possibly remote) directories. It launches instances of itself on each of these machines (by ssh), playing respectively Neil and Oscar's roles.

Communication between Neil and Oscar is handled by the original instance, that links each agent standard output to the standard input of the other.

Niel does almost all computations, while Oscar send him the needed informations and run the effective transfer of files when the computations are done. Btrsync uses rsync to synchronize single files, because it's algorithm to detect changes in a files is very good.

7.2 Time Measurements

Because of difficulties in linking with the GMP library the code is significantly slower than it could be (especially in the computation of the primes from the hashes).

TODO: benchmarks with time + bandwidth (our only benefit ..)

8 Conclusion and Further Improvements

In this work we [to be completed by David]

Mention that the determination of the optimal u is an interesting open question

9 Acknowledgment

The authors acknowledge Guillain Potron for his early involvement in this research work.

todo: Fix euclidean to Euclidean in reference 5.

todo: Merge two reference files rsynch and wagner.

References

1. Burnikel, C., Ziegler, J., Im Stadtwald, D., et al.: Fast recursive division (1998)
2. Fouque, P.A., Stern, J., Wackers, J.G.: Cryptocomputing with rationals. In: Blaze, M. (ed.) Financial Cryptography. Lecture Notes in Computer Science, vol. 2357, pp. 136–146. Springer (2002)
3. Hofmann, M., Pierce, B.C., Wagner, D.: Edit lenses. In: Field, J., Hicks, M. (eds.) POPL. pp. 495–508. ACM (2012)
4. Pan, V., Wang, X.: On rational number reconstruction and approximation. SIAM Journal on Computing 33, 502 (2004)
5. Schönhage, A., Strassen, V.: Schnelle multiplikation grosser zahlen. Computing 7(3), 281–292 (1971)
6. Tridgell, A.: Efficient algorithms for sorting and synchronization. Ph.D. thesis, PhD thesis, The Australian National University (1999)
7. Vallée, B.: Gauss' algorithm revisited. J. Algorithms 12(4), 556–572 (1991)
8. Wang, X., Pan, V.: Acceleration of euclidean algorithm and rational number reconstruction. SIAM Journal on Computing 32(2), 548 (2003)

References

1. Y. Minsky, A. Trachtenberg, *Scalable Set Reconciliation*, 40th Annual Allerton Conference on Communications, Control and Computing, Monticello, IL, October 2002. A full version entitled *Practical Set Reconciliation* can be downloaded from `http://ipsit.bu.edu/documents/BUTR2002-01.ps`
2. Y. Minsky, A. Trachtenberg, R. Zippel, (2003). *Set reconciliation with nearly optimal communication complexity*. IEEE Transactions on Information Theory, 49(9), pp. 2213–2218.
3. Julien Cathalo, David Naccache, Jean-Jacques Quisquater, Comparing with RSA, Cryptography and Coding, Lecture Notes in Computer Science Volume 5921, 2009, pp 326-335, Cryptography and Coding, 12th IMA International Conference, Cryptography and Coding 2009, Cirencester, UK, December 15-17, 2009. Proceedings. Springer-Verlag Berlin Heidelberg
4. D. Eppstein, M. Goodrich, F. Uyeda, G. Varghese What's the difference?: efficient set reconciliation without prior context ACM SIGCOMM Computer Communication Review - SIGCOMM '11, 41(4), pp. 218-229, 2011.

A Extended Protocol

First phase during which Neil amasses modular information on the difference

Oscar		**Neil**
		start protocol with p_1
	$\xrightarrow{c_1}$	
		computes a, b using p_1 if a factors properly then go to Final Phase else perform protocol with p_2
	$\xrightarrow{c_2}$	
		computes $c \bmod p_1 p_2 = \mathrm{CRT}_{p_1,p_2}(c_1, c_2)$ computes a, b using $p_1 p_2$ if a factors properly then go to Final Phase else perform protocol with p_3
	$\xrightarrow{c_3}$	
		computes $c \bmod p_1 p_2 p_3 = \mathrm{CRT}_{p_1,p_2,p_3}(c_1, c_2, c_3)$ computes a, b using $p_1 p_2 p_3$ if a factors properly then go to Final Phase else perform protocol with p_4
	$\vdots$	

Final Phase

		Let $\mathfrak{S} = \{F'_i \text{ s.t. } a \bmod h'_i = 0\}$
	$\xleftarrow{\mathfrak{S},b}$	
deletes files s.t. $b \bmod h_i = 0$ adds $\mathfrak{S}$ to the disk		

Note that the parties do not need to store the p_i's in full. Indeed, the p_i could be subsequent primes sharing their most significant bits. This reduces storage per prime to a small corrected additive constant $\cong \ln(p_i) \cong \ln(2^{2tu+2}) \cong 1.39(tu+1)$ whose storage requires essentially $\log_2(tu)$ bits.

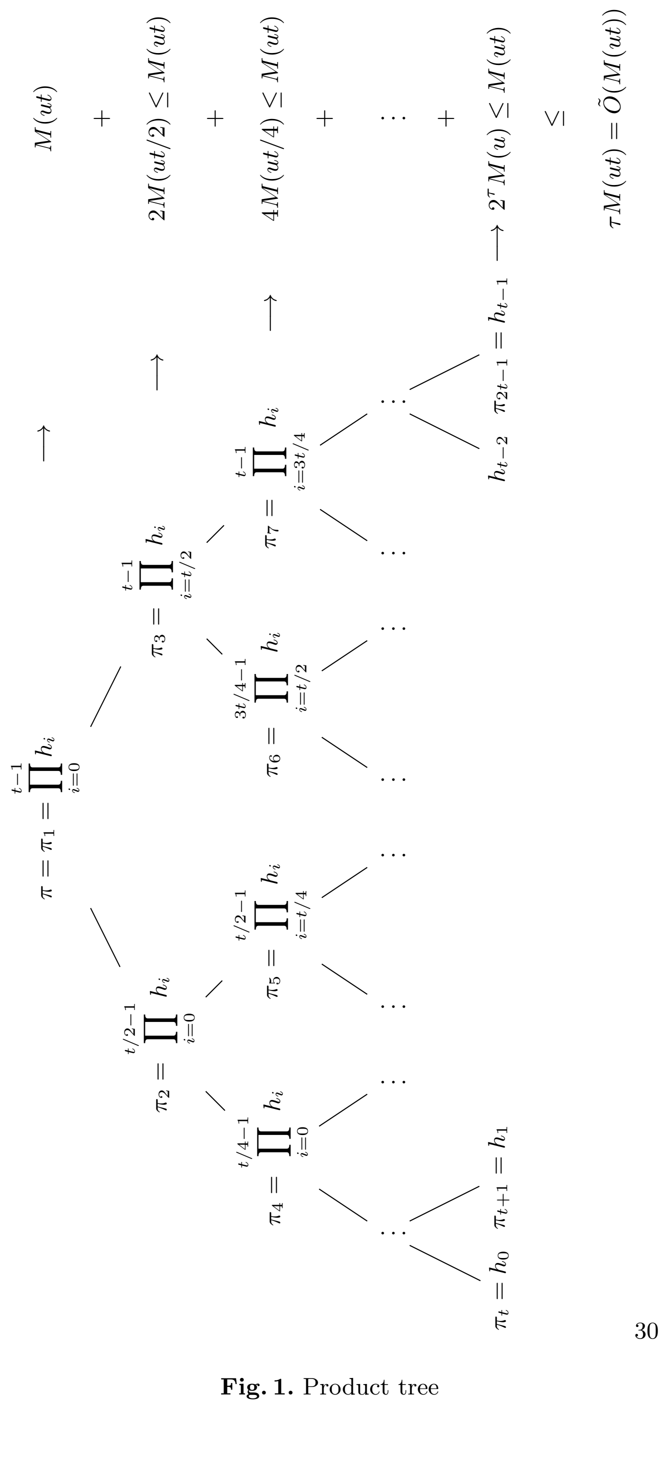
$\pi = \pi_1 = \prod_{i=0}^{t-1} h_i$
$\pi_2 = \prod_{i=0}^{t/2-1} h_i$
$\pi_3 = \prod_{i=t/2}^{t-1} h_i$
$\pi_4 = \prod_{i=0}^{t/4-1} h_i$
$\pi_5 = \prod_{i=t/4}^{t/2-1} h_i$
$\pi_6 = \prod_{i=t/2}^{3t/4-1} h_i$
$\pi_7 = \prod_{i=3t/4}^{t-1} h_i$
$\pi_t = h_0$
$\pi_{t+1} = h_1$
h_{t-2}
$\pi_{2t-1} = h_{t-1}$
$\longrightarrow$ $M(ut)$
$+$
$\longrightarrow$ $2M(ut/2) \leq M(ut)$
$+$
$\longrightarrow$ $4M(ut/4) \leq M(ut)$
$+$
$\cdots$
$+$
$\longrightarrow$ $2^{\tau} M(u) \leq M(ut)$
$\leq$
$\tau M(ut) = \tilde{O}(M(ut))$
30

Fig. 1. Product tree

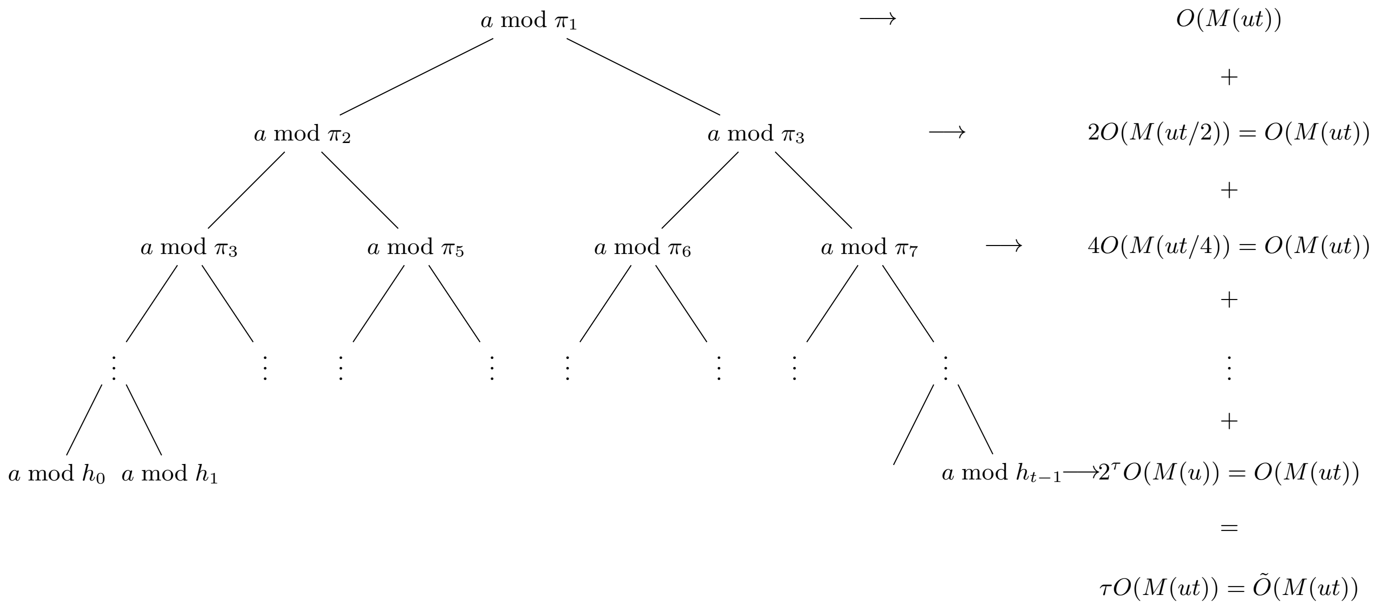

Fig. 2. Division from product tree

Datasets		**Bandwidth (bytes)**						**Time (s)**	
Neil	**Oscar**	**TX$_r$**	**RX$_r$**	**TX$_b$**	**RX$_b$**	**abs**	**rel**	**t$_r$**	**t$_b$**
`source`	`empty`	1613	778353	1846	788357	10237	+2 %	0.2	7.7
`empty`	`source`	11	29	12436	6120	18516	+46305 %	0.1	5.5
`empty`	`empty`	11	29	19	28	7	+32 %	0.1	0.3
`synthetic`	`synthetic_shuffled`	24891	51019	3638	4147	-68125	-57 %	0.2	26.8
`synthetic_shuffled`	`synthetic`	24701	50625	3443	3477	-68406	-58 %	0.2	26.6
`synthetic`	`synthetic`	25011	50918	327	28	-75574	-67 %	0.1	25.7
`firefox-13.0.1`	`firefox-13.0`	90598	28003573	80895	27995969	-17307	+0 %	2.6	4.2
`source_moved`	`source`	2456	694003	1603	1974	-692882	-99 %	0.2	2.5

Fig. 3. Experimental results. The two first columns indicate the datasets, synchronization is performed *from* Neil *to* Oscar. RX and TX are received and sent byte counts, r and b are `rsync` and `btrsync`, we also provide the absolute difference in exchanged data (positive when `btrsync` transfers more data than `rsync`) and the relative amount of data sent by `btrsync` compared to `rsync` (over 100% when `btrsync` transfers more data than `rsync`). The last two columns show timing results.