

When File Synchronization Meets Number Theory

Antoine Amarilli, Fabrice Ben Hamouda, Florian Bourse,
Robin Morisset, 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. This work revisits *set reconciliation*, a problem consisting in synchronizing two multisets while minimizing communication. We propose a new number theoretic reconciliation protocol called “Divide & Factor”. In terms of asymptotic transmission complexity, Divide & Factor is comparable to prior proposals (that anyhow reached optimality). Nonetheless, this new protocol offers interesting parameter trade-offs resulting in experimentally measured *constant*-factor transmission gains over the popular software `rsync`.

Reconciliation experiments show that the new protocol usually transmits less data than `rsync` but requires lengthier calculations.

1 Introduction

This work revisits *set reconciliation*, a problem consisting in synchronizing two multisets while minimizing communication complexity. Set reconciliation is necessary in many practical situations, the most typical of which is certainly incremental information backup.

Set reconciliation has already several efficient and elegant solutions. For instance, [3] presents a particularly interesting reconciliation protocol whose computational and communication complexities are linear in the number of differences between the reconciled multisets.

We refer the reader to [3,4,5] (to quote a few references) for more on the problem’s history and its existing solutions.

This article proposes a new reconciliation protocol based on number theory. In terms of asymptotic transmission complexity, the proposed procedure is comparable to prior proposals (that anyhow reached optimality). Nonetheless, the new protocols offer interesting parameter trade-offs resulting in experimentally measured *constant*-factor gains over the popular software `rsync`.

We observed during most of our reconciliation experiments that the new protocol transmitted less bytes than `rsync` but required lengthier calculations.

Beyond these constant-factor gains, the mathematical ideas underlying this work seem new and interesting as such.

2 “Divide & Factor” Set Reconciliation

2.1 Problem Definition and Notations

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 Neil¹ is that of *exchanging as little data as*

¹ Oscar and Neil will respectively stand for *old* and *new*.

possible during the synchronization process. In practice \mathfrak{D} and \mathfrak{D}' usually differ both in their files and in their tree structure.

In tackling this problem we separate the “what” from the “where” and disregard the relative position of files in subdirectories². 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 $\mathfrak{F}' = \{F'_0, \dots, F'_{n'}\}$.

Let t be the number of discrepancies between \mathfrak{F} and \mathfrak{F}' that Oscar wishes to learn, i.e.:

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

Let $\text{Hash}(F)$ be a collision-resistant hash function³ where F is a file. Let $\text{HashPrime}(F)$ be a function hashing (uniformly) files into primes smaller than 2^u . Define the shorthand notations: $h_i = \text{HashPrime}(F_i)$ and $h'_i = \text{HashPrime}(F'_i)$.

2.2 Description of the Basic Exchanges

We generate a prime p such that:

$$2^{2ut+1} \leq p < 2^{2ut+2} \quad (1)$$

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

$$c = \prod_{F_i \in \mathfrak{F}} \text{HashPrime}(F_i) = \prod_{i=1}^n h_i \bmod p$$

Neil computes:

$$c' = \prod_{F'_i \in \mathfrak{F}'} \text{HashPrime}(F'_i) = \prod_{i=1}^{n'} h'_i \bmod p \quad \text{and} \quad s = \frac{c'}{c} \bmod p$$

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

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

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 a and b can be efficiently recovered from s (A problem known as the *Rational Number Reconstruction* [4,8]). This is typically done using Gauss’ algorithm for finding the shortest vector in a bi-dimensional lattice [7].

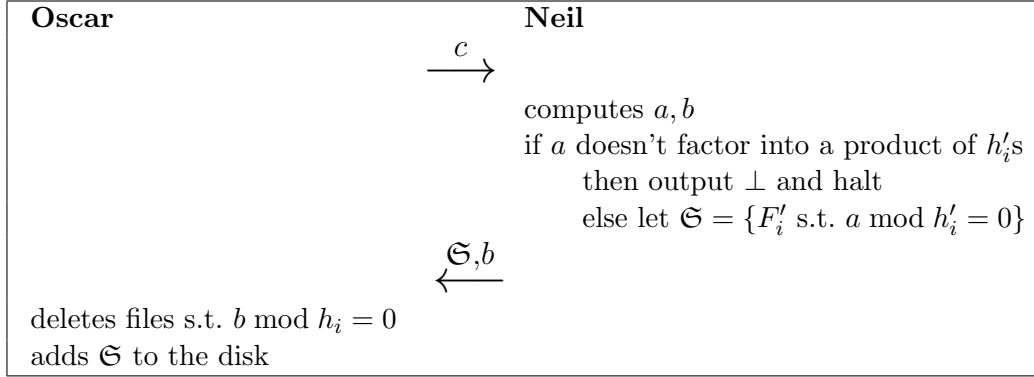
Theorem 1. *Let $a, b \in \mathbb{Z}$ such that $-A \leq a \leq A$ and $0 < b \leq B$. Let $p > 2AB$ be a prime and $s = ab^{-1} \bmod p$. Then a, b can be recovered from A, B, s, p in polynomial time.*

² i.e. view directories as multisets of files.

³ e.g. SHA-1

Taking $A = B = 2^{ut} - 1$, (1) implies that $2AB < p$. Moreover, $0 \leq a \leq A$ and $0 < b \leq B$. 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 deterministically identify the discrepancies between \mathfrak{F} and \mathfrak{F}' and settle them.

Formally, this is done as follows:



As we have just seen, the “output \perp and halt” protocol interruption should never occur if bounds on parameters are respected. However, a file synchronization procedure that works *only* for a limited number of differences is not really useful in practice. Section 2.3 explains how to extend the protocol even when the number of differences exceeds t , the informational capacity of the modulus p .

2.3 The Case of Insufficient Information

To extend the protocol to an arbitrary t , Oscar and Neil agree on an infinite set of primes p_1, p_2, \dots . As long as the protocol fails, Neil will keep accumulating information about the difference between \mathfrak{F} and \mathfrak{F}' as shown in appendix A. Note that no information is lost and that the transmitted modular knowledge about the difference adds-up until it reaches a threshold sufficient to reconcile \mathfrak{F} and \mathfrak{F}' .

3 Efficiency Considerations

This section explores two strategies to reduce the size of p and hence improve transmission by *constant factors* (from an asymptotic communication standpoint, nothing can be done as the protocol already transmits information proportional to t , the difference to settle).

3.1 Probabilistic Decoding: Reducing p

Generate a prime p about twice shorter than the p recommended in section 2.2, namely:

$$2^{ut+w-1} < p \leq 2^{ut+w} \quad (2)$$

where $w \geq 1$ is some small integer (say $w = 50$). Let $\eta = \max(n, n')$. The new redundancy c is calculated as previously and is hence also approximately twice smaller. Namely:

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

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

$$ab \leq 2^{ut} \quad (3)$$

By opposition to section 2.2 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 sequence of at most $\lceil ut/w \rceil + 1$ couples of bounds:

$$(A_i, B_i) = \left(2^{wi}, \left\lfloor \frac{p-1}{2^{wi+1}} \right\rfloor \right) \text{ where } B_i > 1 \text{ and } \forall i > 0, 2A_iB_i < p$$

Equations (2) and (3) imply that 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, b) , and hence the difference between \mathfrak{F} and \mathfrak{F}' .

The problem is that (unlike section 2.2) we have no guarantee that such an (a, b) is unique. Namely, we could (in theory) stumble over an $(a', b') \neq (a, b)$ satisfying (3) for some index $i' \neq i$. We expect this to happen with negligible probability (that we do not try to estimate here) when w is large enough, but this makes the modified protocol heuristic only.

3.2 The File Laundry: Reducing u

What happens if we brutally shorten u in the basic Divide & Factor protocol?

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

Consider **HashPrime** as a random function from $\{0, 1\}^*$ to $\{0, \dots, 2^u - 1\}$. Let X_i be the random variable:

$$X_i = \begin{cases} 1 & \text{if file } F_i \text{ collides with another file.} \\ 0 & \text{otherwise.} \end{cases}$$

Clearly, we have $\Pr[X_i = 1] \leq \frac{\eta-1}{2^u}$. The average number of colliding files 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.

However, it is important to note that a collision can only yield a *false positive*, and never a *false negative*. In other words, while a collision may obviate a difference⁴ a collision will never create a nonexistent difference *ex nihilo*.

⁴ e.g. make the parties blind to the difference between `index.htm` and `iexplore.exe`.

Thus, it suffices to replace $\text{HashPrime}(F)$ by a diversified $\text{HashPrime}(k|F)$ to quickly filter-out file differences by repeating the protocol for $k = 1, 2, \dots$. At each iteration the parties will detect new files and new deletions, fix these and “launder” again the remaining multisets.

Assume that diversified $\text{HashPrime}(k|F)$ ’s are random and independent. To understand why the probability that a stubborn file persists colliding decreases exponentially with the number of iterations k let us assume that η remains invariant between iterations and define the following random variables:

$$X_i^\ell = \begin{cases} 1 & \text{if file } F_i \text{ collides with another file during iteration } \ell. \\ 0 & \text{otherwise.} \end{cases}$$

$$Y_i = \prod_{\ell=1}^k X_i^\ell = \begin{cases} 1 & \text{if file } F_i \text{ collides with another file during the } k \text{ first protocol iterations.} \\ 0 & \text{otherwise.} \end{cases}$$

By independence, we have:

$$\Pr[Y_i = 1] = \prod_{\ell=1}^k \Pr[X_i^\ell = 1] = \Pr[X_i^1 = 1] \dots \Pr[X_i^k = 1] \leq \left(\frac{\eta - 1}{2^u}\right)^k$$

Therefore the average number of colliding files is:

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

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

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

For the previously considered instance⁵ we get $\epsilon_2 \leq 5.43\%$ and $\epsilon_3 \leq 2 \cdot 10^{-3}\%$.

A more refined (but somewhat technical) analysis As mentioned previously, the parties can remove the files confirmed as different during iteration k and work during iteration $k + 1$ only with common and colliding files. Now, the only collisions that can fool round k , are the collisions of a file-pairs (F_i, F_j) such that F_i and F_j have both already collided during *all the previous iterations*⁶. We call such collisions “masquerade balls”. Define the random variable:

$$Z_i^\ell = \begin{cases} 1 & \text{if } F_i \text{ participated in masquerade balls during all } \ell \text{ first protocol iterations.} \\ 0 & \text{otherwise.} \end{cases}$$

*Inclure ici image du
bal masque*

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 $k \geq 1$, we have:

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

Fabrice tu utilises la notation non définie $X_{i,j}^k$ peux-tu la préciser stp?

Furthermore $p_0 = 1$ and

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

hence:

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

and

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

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

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

For $(\eta = 10^6, u = 32, k = 2)$, we get $\epsilon'_2 \leq 0.013\%$.

⁵ $\eta = 10^6, u = 32$.

⁶ Note that we do not require that F_i and F_j repeatedly collide *with each other*. e.g. we may witness during the first round $h_{1,u}(F_1) = h_{1,u}(F_2)$ and $h_{1,u}(F_3) = h_{1,u}(F_4)$ while during the second round $h_{2,u}(F_1) = h_{2,u}(F_4)$ and $h_{2,u}(F_2) = h_{2,u}(F_3)$.

How to select u ? For a fixed k , ϵ'_k decreases as u grows. For a fixed u , ϵ'_k also decreases as k grows. Transmission, however, grows with both u (bigger digests) and k (more iterations). We write for the sake of clarity: $\epsilon'_k = \epsilon'_{k,u,\eta}$.

Fix η . Note that the number of bits transmitted per iteration ($\simeq 3ut$), is proportional to u . This yields an expected transmission complexity bound $T_{u,\eta}$ such that:

$$T_{u,\eta} \propto u \sum_{k=1}^{\infty} k \cdot \epsilon'_{k,u,\eta} = \frac{u\eta(\eta-1)}{2^u} \sum_{k=1}^{\infty} k \left(\frac{1}{2^u} + \frac{(\eta-2)^2}{2^{2u}} \right)^{k-1} = \frac{u\eta(\eta-1)8^u}{(2^u - 4^u + (\eta-2)^2)^2}$$

Dropping the proportionality factor $\eta(\eta-1)$, neglecting $2^u \ll 2^{2u}$ and approximating $(\eta-2) \simeq \eta$, we can optimize the function:

$$\phi_{\eta}(u) = \frac{u \cdot 8^u}{(4^u - \eta^2)^2}$$

$\phi_{106}(u)$ admits an optimum for $u = 19$.

Note: The previous analysis is a rough approximation, in particular:

- We consider u -bit prime digests while u -bit strings contain only about $2^u/u$ primes.
- In all our probability calculations η can be replaced by the total number of differences t . It is reasonable to assume that in most *practical* settings $t \ll \eta$, but extreme instances where $t \sim \eta$ can sometimes be encountered as well.
- We used a fixed u in all rounds. Nothing forbids using a different u_k at each iteration⁷.
- Our analysis treated t as a constant, but large t values increase p and hence the number of potential files detected as different per iteration - an effect disregarded in our analysis.

Given that, after all, optimization may only result in constant-factor improvements, we suggest to optimize t and u experimentally, e.g. using the open source program **btrsync** developed by the authors (cf. section 5).

3.3 How to Stop a Probabilistic Washing Machine?

We now combine both optimizations and assume that ℓ laundry rounds are necessary for completing some given reconciliation task using a shortened p . Unlike section 2.2, confirming correct protocol termination is now non-trivial.

Let the round failure probability⁸ be some function $v(w)$ (that we did not estimate). If w is kept small (for efficiency reasons), the probability $(1 - v(w))^\ell$ that the protocol will properly terminate may dangerously drift away from one.

If v of $\ell + v$ rounds fail, Oscar needs to solve a problem called *Chinese Remaindering With Errors*:

⁷ ...or even fine-tuning the u_k s adaptively, as a function of the laundry's effect on the progressively reconciliated multisets.

⁸ i.e. that probability that a round resulted in an $(a', b') \neq (a, b)$ satisfying equation (3).

Problem 1. (Chinese Remaindering With Errors: CRWE). Given as input integers v , B and $\ell+v$ points $(s_1, p_1), \dots, (s_{\ell+v}, p_{\ell+v}) \in \mathbb{N}^2$ where the p_i 's are coprime, output all numbers $0 \leq s < B$ such that $s \equiv s_i \pmod{p_i}$ for at least v values of i .

We refer the reader to [2] for more on this problem, which is beyond the scope of this article and note that Boneh [1] provides a polynomial-time algorithm for solving the CRWE problem under certain conditions satisfied by our setting.

To detect that reconciliation succeeded, Neil will send to Oscar $\text{Hash}(\mathfrak{F}')$ as soon as the interaction starts. As long as Oscar's CRWE resolution does not result in a state matching $\text{Hash}(\mathfrak{F}')$, the parties will continue the interaction.

4 Asymptotic Computational Complexity

Let $\mu(k)$ be the time required to multiply two k -bit numbers⁹. The modular division of two k -bit numbers and the reduction of $2k$ -bit number modulo a k -bit number are known to cost $\tilde{O}(\mu(k))$ [1].

For naïve (i.e. convolutive) algorithms $\mu(k) = O(k^2)$, but using FFT multiplication strategies [5], $\mu(k) = \tilde{O}(k)$. FFT is experimentally faster than convolutive methods from $k \sim 10^6$ and on. For such sizes, in packages such as **gmp**, division and modular reduction also run in $\tilde{O}(\mu(k))$. For $p \sim 2^{ut}$ we get the following complexity analysis:

Entity	Computation	Complexity expressed in \tilde{O} of			
Both	redundancies c and c'	$n \cdot \mu(ut)$	naïve product	nut	using FFT
Oscar	$s = c'/c \pmod{p}$	$\mu(ut)$	naïve inversion	ut	using FFT
Oscar	a, b such that $s = a/b \pmod{p}$	$(ut)^2$	naïve ext. GCD	$\mu(ut)$	using [4,8]
Both	factorization of a (resp. b) by modular reductions	$n \cdot \mu(ut)$	naïve reduction	nut	using FFT
	Overwhelming complexity:	$\max((ut)^2, n \cdot \mu(ut))$		nut	

Fig. 1. Global Protocol Complexity.

4.1 Improvements

It is possible to reduce the overwhelming complexities of the computation of c, c' and the factorization to $\tilde{O}(n/t\mu(ut))$ using convolutive methods and to $\tilde{O}(nu)$ with FFT [5]. To simplify explanations, assume that t is a power of two and $t = 2^\tau$ divides n .

The idea is the following: we group h_i by group of t elements and compute the product of each such group in \mathbb{N} .

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

⁹ We suppose that $\forall k, k', \mu(k + k') \geq \mu(k) + \mu(k')$

Each H_j can be computed in $\tilde{O}(\mu(ut))$ using the standard product tree method given in Algorithm 1 (for $j = 0$) and shown in Figure 2. And all these n/t products can be computed in $\tilde{O}(n/t\mu(ut))$. Then, one can compute c by multiplying the H_j modulo p , which costs $\tilde{O}(n/t\mu(ut))$.

Algorithm 1 Product Tree Algorithm

Require: a table h such that $h[i] = h_i$
Ensure: $\pi = \pi_1 = \prod_{i=0}^{t-1} h_i$, and $\pi[i] = \pi_i$ for $i \in \{1, \dots, 2t-1\}$ as in Figure 2

```

1:  $\pi \leftarrow$  array of size  $t$ 
2: function PRODTREE( $i, \text{start}, \text{end}$ )
3:   if  $\text{start} = \text{end}$  then
4:     return 1
5:   else if  $\text{start} + 1 = \text{end}$  then
6:     return  $h[\text{start}]$ 
7:   else
8:      $\text{mid} \leftarrow \lfloor (\text{start} + \text{end}) / 2 \rfloor$ 
9:      $\pi[i] \leftarrow \text{PRODTREE}(2 \times i, \text{start}, \text{mid})$ 
10:     $\pi[i + 1] \leftarrow \text{PRODTREE}(2 \times i + 1, \text{start}, \text{mid})$ 
11:    return  $\times \text{PRODTREE}(\text{mid}, \text{end})$ 
12:  $\pi[1] \leftarrow \text{PRODTREE}(1, 0, t)$ 
```

The same technique applies to factorization¹⁰ with a slight *caveat*.

After computing the tree product we can compute the residues of a modulo H_j . Then we can compute the residues of $a \bmod H_j$ modulo the two children of H_j in the product tree (Namely: $\prod_{i=jt}^{jt+t/2-1} h_i$ and $\prod_{i=jt}^{jt+t/2-1} h_i$). Then we can compute the residues of these two values modulo the children of the previous children, and so on. Intuitively, we descend 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 is illustrated in Algorithm 3 and Figure 3 (for $j = 1$). Complexity is $\tilde{O}(\mu(ut))$, for each j summing-up to a total complexity of $\tilde{O}(n/t\tilde{O}(\mu(ut)))$.

¹⁰ We illustrate the process with a , this is applicable *ne variatur* to b as well.

Algorithm 2 Division Using a Product Tree

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

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

```

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

4.2 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'intérêt 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.

Fabrice: Je comprends que ca marche si p est le produit de p_i immediatement superieurs a 2^u . Je ne vois pas pourquoi on ne pourrait pas aller encore plus loin et prendre carrement un $p = 2 \times 3 \times 5 \times \dots$ FRIABLE. Qu'en penses-tu? Il me semble que toutes les maths du paper continuent a etre valables non?

5 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 `btrsycn`, can be downloaded from [6].

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

5.1 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.

5.2 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 their 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 coming 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 other 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...

5.3 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 $a \rightarrow b$, 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 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 moves 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.

An optimization implemented by **btrfsync** 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 as they don't need to copy the file contents.

5.4 Experimental Comparison to **rsync**

We compared **rsync**¹¹ and our Divide & Factor implementation called **btrfsync** under the following experimental conditions:

Test Directories: The directories used for transmission and time comparisons are described in Figure 4.

Test Directories: **rsync** was called with the following flag configurations, for the following reasons:

- ▶ **--delete** to delete existing files on Oscar which do not exist on Neil.
- ▶ **-I** to ensure that **rsync** did not cheat by looking at the file modification times (which **btrfsync** does not do).
- ▶ **--chmod="a=rx,u+w"** in an attempt to disable the transfer of file permissions (which **btrfsync** 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 **btrfsync** for this reason.
- ▶ **-v** Transmission accounting was performed by calling **rsync** with the **-v** flag (which reports the number of sent and received bytes). For **btrfsync** we added a piece of code counting the amount of data transmitted during **btrfsync**'s own negotiations.

Network Configuration: The experiments were performed between two remote hosts connected by a high-speed link. Time measurements account both for CPU time and for transfer times and are hence only given as a general indication.

¹¹ version 3.0.9, used both as a competitor to benchmark against and for a minor underlying call to **rsync** in our own code.

Algorithm 3 Perform Moves

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

```

1:  $M \leftarrow []$ 
2:  $T \leftarrow []$ 
3: for  $f$  in  $D$ 's keys do
4:    $M[f] \leftarrow \text{not\_done}$ 
5: function UNBLOCK_COPY( $f, t$ )
6:   if  $t$  is blocked by some  $b$  then
7:     if  $b$  is not in  $D$ 's keys then
8:       unlink( $b$ )
9:     else
10:      RESOLVE( $b$ )
11:   if  $T[f]$  was set then
12:      $f \leftarrow T[f]$ 
13:   copy( $f, d$ )
14: function RESOLVE( $f$ )
15:   if  $M[f] = \text{done}$  then
16:     return
17:   if  $M[f] = \text{doing}$  then
18:      $T[f] \leftarrow \text{mktemp}()$ 
19:     move( $f, T[f]$ )
20:      $M[f] \leftarrow \text{done}$ 
21:     return
22:    $M[f] \leftarrow \text{doing}$ 
23:   for  $d \in D[f]$  do
24:     if  $d \neq f$  then
25:       unblock_copy( $f, d$ )
26:   if  $f \notin D[f]$  and  $T[f]$  was not set then
27:     unlink( $f$ )
28:   if  $T[f]$  was set then
29:     unlink( $T[f]$ )
30: for  $f$  in  $D$ 's keys do
31:   RESOLVE( $f$ )

```

▷ We don't need b

▷ Take care of b and make it go away

▷ Already managed by another in-edge

▷ We found a loop, moved f out of the way

▷ Perform all the moves

5.5 Results:

Results are given in table 5. In general, **btrsyc** spends more time than **rsync** on computations (especially when the number of files is large, which is typically seen with our **synthetic** experiments).

The transmission results, however, turn out to be favorable to **btrsyc**:

In the trivial experiments where either Oscar or Neil have no data at all, **rsync** outperforms **btrsyc**. This is especially clear when Neil has no data: **rsync** immediately notices that there is nothing to transfer, but **btrsyc** starts information transfers to determine the symmetric difference.

On the non-trivial datasets, however, **btrsyc** outperforms **rsync**. This is the case on the **synthetic** datasets, where **btrsyc** 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. In the case of Firefox source code datasets, **btrsyc** saves a very small amount of bandwidth, presumably because of unmodified files. For the **btrsyc** source code dataset, we notice that **btrsyc**, unlike **rsync**, was able to detect the move and avoid retransferring the moved folder.

Code Availability: Source codes can be downloaded from <https://github.com/RobinMorisset/Btrsyc>..

6 Haskell Implementation

6.1 Program Structure

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

A proof-of-concept called Btrsyc has been implemented in Haskell and is available at

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.

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

7 Conclusion and Further Improvements

We strongly encourage the developer community to continue improving our open source and public-domain software (that we called **btrsyc**).

Recopier de l’intro et adapter

8 Acknowledgment

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

9 ToDo

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. D. Boneh, *Finding Smooth Integers in Short Intervals Using CRT Decoding*, Proceedings of the 32-nd Annual ACM Symposium on Theory of Computing, 2000, pp. 265–272.
2. D. Bleichenbacher and Ph. Nguyen, *Advances in Cryptology – Proceedings of Eurocrypt’00*, vol. 1807 of Lecture Notes in Computer Science, Springer-Verlag, pp. 53–69.
3. 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>
4. Y. Minsky, A. Trachtenberg, R. Zippel, *Set reconciliation with nearly optimal communication complexity*. IEEE Transactions on Information Theory, 49(9), 2003, pp. 2213–2218.
5. 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), 2011, pp. 218-229.
6. <https://github.com/RobinMorisset/Btrsynchron>

A Extended Protocol

First phase during which Neil amasses modular information on the difference	
Oscar	Neil
	start the protocol with p_1
$\xrightarrow{c_1}$	computes a, b using p_1 if a factors properly then go to Final Phase else perform the protocol with p_2
$\xrightarrow{c_2}$	computes $c \bmod p_1 p_2 = \text{CRT}_{p_1, p_2}(c_1, c_2)$ computes a, b using $p_1 p_2$ if a factors properly then go to Final Phase else perform the protocol with p_3
$\xrightarrow{c_3}$	computes $c \bmod p_1 p_2 p_3 = \text{CRT}_{p_1, p_2, p_3}(c_1, c_2, c_3)$ computes a, b using $p_1 p_2 p_3$ if a factors properly then go to Final Phase else perform the 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 parties do not need to store the p_i 's in full. Indeed, the p_i s could be subsequent primes sharing their most significant bits. This reduces storage per prime to a very small additive constant $\cong \ln(p_i) \cong \ln(2^{2tu+2}) \cong 1.39(tu + 1)$ of about $\log_2(tu)$ bits.

B Hashing Into Primes

Hashing into primes is frequently needed in cryptography. A recommended implementation of $\text{HashPrime}(F)$ is typically algorithm 5. If u is large enough (e.g. 160) one might sacrifice uniformity to avoid repeated file hashings and use $\text{HashPrime}(F) = \text{NextPrime}(\text{Hash}(F))$. Yet another acceleration option (even less uniform) consists in replacing NextPrime by the faster nonuniform scanning Algorithm 4 where $\alpha = 2 \times 3 \times 5 \times \cdots \times q$ is the product of the first primes until some moderate bound.

Algorithm 4 Fast Nonuniform Hashing Into Primes

```

1:  $h = \alpha \left\lfloor \frac{\text{Hash}(F)}{\alpha} \right\rfloor + 1$ 
2: while  $h$  is composite do
3:    $h = h - \alpha$ 
4: return  $h$ 

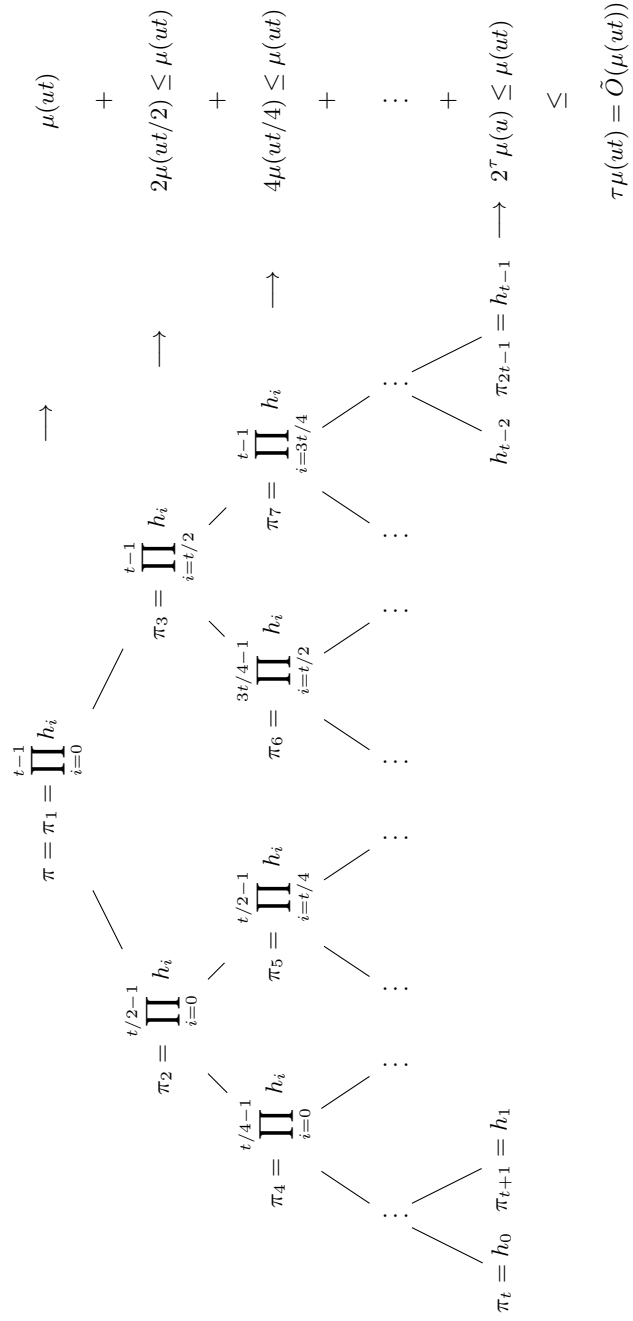
```

Algorithm 5 Possible Implementation of HashPrime(F)

```

1:  $i = 0$ 
2: repeat
3:    $h = 2 \cdot \text{Hash}(F|i) + 1$ 
4:    $i = i + 1$ 
5: until  $h$  is prime
6: return  $h$ 

```

**Fig. 2.** Product tree

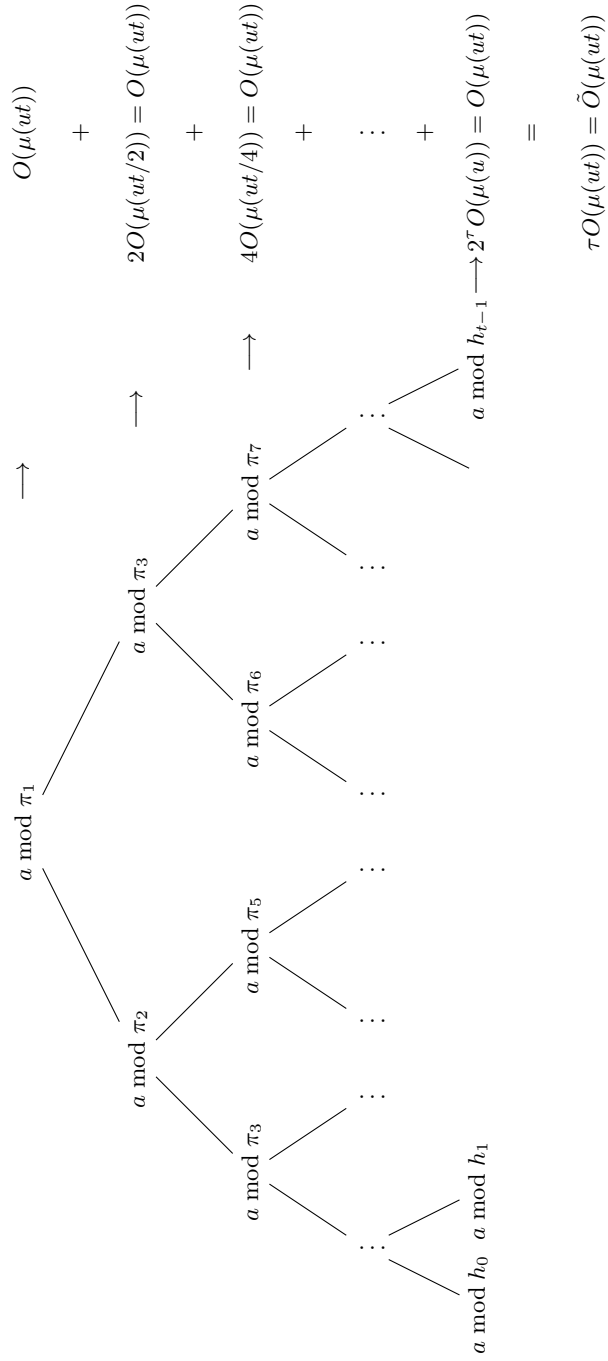


Fig. 3. Division From Product Tree

Directory	Description
synthetic	A directory containing 1000 very small files containing the numbers $1, 2, \dots, 1000$.
synthetic_shuffled	synthetic with: 10 deleted files 10 renamed files 10 modified files
source	A snapshot of btrsync 's own source tree
source_moved	source with one big folder (a few megabits) renamed.
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.

Fig. 4. Test Directories.

Entities and Datasets		Transmission (Bytes)						Time (s)	
Neil's \mathfrak{F}'	Oscar's \mathfrak{F}	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. 5. 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.