From Rational Number Reconstruction to Set Reconciliation

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)

Penser à voir si on change le titre.

Abstract. This work revisits *set reconciliation*, a problem consisting in synchronizing two multisets of fixed-size values while minimizing the amount of data transmitted. We propose a new reconciliation protocol called "Divide & Factor" (D&F), based on number theory, which achieves optimal asymptotic transmission complexity like prior proposals. We then study the problem of synchronizing sets of variable-size files, and describe how constant-factor improvements can be achieved through the use of hashing with a carefully chosen hash size (balancing the quantity of data transferred and the risk of collisions). We show how this process can be applied to synchronize file hierarchies, taking into account the location of files. We describe `btrsync`, our open-source implementation of the protocol, and benchmark it against the popular software `rsync` to demonstrate that `btrsync` transmits much less data at the expense of a small computational overhead.

Notations

Cette section est temporaire et ne sert que pour assurer des notations cohérentes le long du papier, notamment pour les indices:

- i, j: pour les fichiers/hashés h_i,F_i
- k: pour les rounds avec changement de modulo: $p_k, c_k, s_k, C_k, S_k, t_k, T_k$
- ℓ: pour les rounds liés aux traitements des collisions
- λ: pour une borne sur le ℓ
- κ: pour une borne pour le k
- $T_k = \sum_{j=1}^{k} t_j$, T le nombre réel de différences, t le nombre maximal accepté au cours d'un round, lorsqu'on ne présente qu'un round.

TODO: Corriger les n,n',η,η'.
TODO: faire commencer le i à 1 PARTOUT

1 Introduction

File synchronization is the important practical problem consisting of retrieving a hierarchy of files on a remote host given some outdated or incomplete version of this hierarchy of files on the local machine. In many use cases, the bottleneck is the bandwidth of the network link between the local machine and the remote host, and care must be taken to limit the

quantity of data transferred over the link by using the existing copy of the set of files to the fullest possible extent. Popular file synchronization programs such as `rsync` use rolling checksums to skip remote file parts which match a file part at the same location on the local machine; however, they are usually unable to take advantage of the local files in subtle ways, like detecting that some large file is already present on the local machine but at a different location.

File synchronization is closely linked to the theoretical problem of *set reconciliation*: given two sets of fixed-size data items on different machines, determine the symmetric difference of the two sets while minimizing the amount of data transferred. The lower bound of the quantity of data to transfer is clearly the size of the symmetric difference (i.e., the number of elements in the difference times the size of these elements), and some known algorithms achieve this bound [6]. We refer the reader to [6,7,3] (to quote a few references) for more on this problem's history and its existing solutions.

In this paper, we look at the problems of set reconciliation and file synchronization from a theoretical and practical perspective. Our contribution are as follows:

- We introduce "Divide & Factor", a new set reconciliation algorithm. D&F is based on number theory: it represents the items to synchronize as prime numbers, accumulates information about the symmetric difference in a series of rounds through the Chinese remainder theorem (CRT), and reconstitutes the result through the use of rational number reconstruction. The algorithm is described in Section 2.
- We show that D&F, like existing algorithms, has a transmission complexity which is linear in the size of the symmetric difference, and discuss a possible constant factor optimization (Section 3).
- We explain in Section 4 how D&F can be extended with hash functions to reconcile sets of files which do not have a fixed size, and analyze how the hash functions should be chosen to achieve a good tradeoff between the quantity of data to transfer and the risks of confusion. Some points of this analysis apply no matter the choice of the underlying set reconciliation algorithm.
- We study the computational complexity of D&F and present possible trade-offs between constant-factor transmission complexity and computational complexity through alternative design choices (Section 5).
- We spell out in Section 6 how the previous construction can be extended to perform file synchronization, taking into account the location and metadata of files and managing situations such as file moves in an intelligent manner. We describe an algorithm to apply a set of move operations on a set of files in-place which avoids excessive use of temporary file locations.
- We present `btrsync`, our implementation of file synchronization through D&F, in Section 7, and benchmark it against `rsync`. The results show that `btrsync` has a higher computational complexity but transmits less data in most scenarios.

2 "Divide & Factor" Set Reconciliation

This section shows our new set reconciliation protocol D&F for set two of u-bit primes $\mathcal{H}$ and $\mathcal{H}'$, which is based on number theory. After introducing the problem and the notations, we first present a basic version of D&F for a bounded number of differences between the two sets. Then we show how to extend this protocol to deal with any number of differences.

2.1 Problem Definition and Notations

Oscar possesses an old version of a multiset $\mathcal{H}$ of n u-bit primes $\mathcal{H}$ that he wished to update. Neil has the new, up-to-date version of this multiset, denoted $\mathcal{H}'$ and containing n' u-bit primes.

Fabrice: should we speak about the fact we are limited to primes ???

Let us write $\mathcal{H} = \{h_1, \ldots, h_n\}$ and $\mathcal{H}' = \{h'_1, \ldots, h'_n\}$, such that: $h_1 < h_2 < \cdots < h_n$ and $h'_1 < h'_2 < \cdots < h'_{n'}$. Let $\mathcal{D} = \mathcal{H} \setminus \mathcal{H}'$ be the numbers deleted in Neil's version (compared to Oscar's version), and $\mathcal{D}' = \mathcal{H}' \setminus \mathcal{H}$ be the numbers added in Neil's version. Let T be the number of discrepancies or differences between $\mathcal{H}$ and $\mathcal{H}'$:

$$T = \#\mathcal{D} + \#\mathcal{D}' = \#\mathcal{H} + \#\mathcal{H}' - 2\#\left(\mathcal{H} \bigcap \mathcal{H}'\right) = \#\left(\mathcal{H} \bigcup \mathcal{H}'\right) - \#\left(\mathcal{H} \bigcap \mathcal{H}'\right),$$

Let also suppose we have a collision-resistant hash function `Hash` which takes as input any bit string and output a short binary string of, for example, 160-bits[1].

2.2 Basic Protocol with Bounded T

In this section, we present a basic version of D&F when the number of differences T is at most t, with t a known constant.

The protocol works as follows. We first generate a prime p such that

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

Given $\mathcal{H}$, Oscar generates and sends to Neil the redundancy:

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

Neil computes:

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

Since $T \leq t$, $\mathcal{H}$ and $\mathcal{H}'$ differ by at most t elements and so s can be written

$$s = \frac{a}{b} \bmod p \text{ where } \begin{cases} a = \prod_{h'_i \in \mathcal{H}' \setminus \mathcal{H}} h'_i \leq 2^{ut} - 1 \\ b = \prod_{h_i \in Set \setminus \mathcal{H}'} h_i \leq 2^{ut} - 1 \end{cases}.$$

[1] TODO: comment on such hash function for Antoine

The problem of recovering a and b from s efficiently is known as *Rational Number Reconstruction* [8,12]. And the following theorem, which is a slightly modified version of Theorem 1 in [4], guarantees that it can be solved efficiently in this setting.

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

Taking $A = B = 2^{ut} - 1$, Equation (1) implies that $AB < p$. Moreover, $0 \leq a \leq A$ and $0 < b \leq B$. Thus Oscar can recover a and b from s in polynomial time: a possible option is to use Gauss algorithm for finding the shortest vector in a bi-dimensional lattice [11]. By testing the divisibility of a and b by the h_i and the h'_i, Neil and Oscar can attempt to identify the discrepancies between $\mathcal{H}$ and $\mathcal{H}'$ and settle them[2].

Fabrice: certes, mais on utilise directement un Euclide étendu tronqué. Et pourquoi citer Vallée qui est un peu incompréhensible dans notre cas... TODO: check that in our program we ensure a and b co-prime !! otherwise, it may fail !!!!!

The basic protocol is depicted in Figure 1.

Oscar		**Neil**
$c \leftarrow \prod_{i=1}^{n} h_i \bmod p$		$c' \leftarrow \prod_{i=1}^{n'} h'_i \bmod p$
	$\xrightarrow{\quad c \quad}$	
		$s \leftarrow c'/c \bmod p$
		reconstruct a, b from s
		$\mathcal{D}' \leftarrow \{h'_i \in \mathcal{H}' \mid a \bmod h'_i = 0\}$
	$\xleftarrow{\quad \mathcal{D}', b \quad}$	
$\mathcal{D} \leftarrow \{h_i \in \mathcal{H} \mid b \bmod h_i\}$		

Fig. 1. Basic D&F Protocol (when $T \leq t$).

2.3 Handling an Unbounded Number of Differences

In practice, we often do not know any reasonable bound t on the number of differences T. That is why, in this section, we extend our protocol to work with any T. We do this in two steps: we first show that we can slightly change our basic protocol to detect when its output is incorrect, when $t < T$. Then we construct a protocol which works with any T.

Detecting Bad Reconciliation We first remark, that if we execute the previous protocol when $t < T$, either a and b are still correct (in which case, the protocol works correctly), or a and b are not correct. In the second case, with high probability, the product of the h_i's which divide a is not equal to a (TODO: prove this claim). This check has the advantage

[2] Actually, this only works if $\mathcal{H}$ and $\mathcal{H}'$ are sets, but if they are multi-set, and if, for example, h'_i is j' times in $\mathcal{H}'$, we need to check the divisibility of b by $h_i, h_i^2, \ldots, h_i^{j'}$. In the sequel, we only present algorithms for $\mathcal{H}$ and $\mathcal{H}'$ sets. We let the reader adapt them to the multi-set case.

to be very fast and not needing Neil to send any data to Oscar. But, if we are unlucky, it can happen that this check is not sufficient. Let us call $\perp$ the event that this check failed.

That is why, we need to add another (more costly) check which can detect any bad reconciliation. This second check is very simple: at the beginning, before the first flow, Neil sends to Oscar a hash of its sets: $H = \texttt{Hash}(\mathcal{H}') = \texttt{Hash}((h'_1, \ldots, h'_{n'}))$, and at the end, after computing $\mathcal{D}$, Oscar computes $\mathcal{H}'$ from $\mathcal{H}$, $\mathcal{D}'$ and $\mathcal{D}$ and check the hash of this new set if H.

Complete D&F Protocol To extend the protocol to an arbitrary T, Oscar and Neil agree on an infinite set of primes $p_1, p_2, \ldots$ As long as the protocol fails (*i.e.*, yields a bad reconciliation), Neil and Oscar redo the protocol with a new p_ℓ and Neil will keep accumulating information about the difference between $\mathcal{H}$ and $\mathcal{H}'$. Each of this repetition is called a round.

More precisely, let us suppose $2^{2ut_k} \leq p_k < 2^{2ut_k+1}$. Let us write $P_k = p_1 \ldots p_k$ and $T_k = t_1 + \cdots + t_k$. After receiving the redundancies $c_1, \ldots, c_k$ corresponding to $p_1, \ldots, p_k$, Neil has as many information as if Oscar had transmitted a redundancy C_k corresponding to the modulo P_k, and can compute $S_k = C'_k/C_k$ from $s_k = c'_k/c_k$ and S_{k-1} using the CRT (TODO ref ?):

$$S_k = \mathrm{CRT}(S_{k-1}, P_{k-1}, s_k, p_k) = S_{k-1}(p_k^{-1} \bmod P_{k-1})p_k + s_k(P_{k-1}^{-1} \bmod p_k)P_{k-1} \bmod P_k.$$

The full protocol is depicted in Figure 2. 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 $\mathcal{H}$ and $\mathcal{H}'$. Therefore, the number κ of rounds used is the minimum number k such that $T_k \geq T$.

In the sequel, we focus on two variants:

- $t_1 = t_2 = \cdots = t$, in which case $\kappa = \lceil T/t \rceil$;
- $t_k = 2^k t'$, in which case $\kappa = \lceil \log_2(T/t') \rceil$.

3 Transmission Complexity

This section proves that D&F achieves optimal asymptotic transmission complexity and explores two strategies for reducing the size of p and hence improving transmission by *constant factors*.

3.1 Proof of Transmission Complexity Optimality

Assuming that there is no event $\perp$ (since these events happen with negligible probability), the transmission complexity of D&F is:

$$\sum_{k=1}^{\kappa} \log c_k + \log b + \log |\mathcal{D}'| \;\leq\; \sum_{k=1}^{\kappa} (ut_k + 1) + \frac{1}{2}(uT_\kappa) + uT \;\leq\; u(T_\kappa + \kappa) + \frac{1}{2}uT_\kappa + uT \;\leq\; \frac{5}{2}uT_\kappa + u\kappa,$$

Oscar		**Neil**
Phase 0: Hash of content for detection of bad reconciliation		
		$H \leftarrow \texttt{Hash}(\mathcal{H}')$
	$\xleftarrow{H}$	
Phase 1: Neil amasses modular information on the difference		
Round 1		
$c_1 \leftarrow \prod_{i=1}^{n} h_i \bmod p_1$		$c'_1 \leftarrow \prod_{i=1}^{n'} h'_i \bmod p_1$
	$\xrightarrow{c_1}$	
		$s_1 \leftarrow c'_1/c_1 \bmod p_1$
		$S_1 \leftarrow s_1$
		reconstruct a, b from S_1 (modulo P_1)
		$\mathcal{D}' \leftarrow \{h'_i \in \mathcal{H}' \mid a \bmod h'_i = 0\}$
		if $\prod_{h \in \mathcal{D}'} h \bmod P_1 = a$ then go to final phase
Round 2		
$c_2 \leftarrow \prod_{i=1}^{n} h_i \bmod p_2$		$c'_2 \leftarrow \prod_{i=1}^{n'} h'_i \bmod p_2$
	$\xrightarrow{c_2}$	
		$s_2 \leftarrow c'_2/c_2 \bmod p_2$
		$S_2 \leftarrow \mathrm{CRT}(S_1, P_1, s_2, p_2)$
		reconstruct a, b from S_2 (modulo P_2)
		$\mathcal{D}' \leftarrow \{h'_i \in \mathcal{H}' \mid a \bmod h'_i = 0\}$
		if $\prod_{h \in \mathcal{D}'} h \bmod P_2 = a$ then go to final phase
Round 3		
$c_3 \leftarrow \prod_{i=1}^{n} h_i \bmod p_3$		$c'_3 \leftarrow \prod_{i=1}^{n'} h'_i \bmod p_3$
	$\xrightarrow{c_3}$	
		$s_3 \leftarrow c'_3/c_3 \bmod p_3$
		$S_3 \leftarrow \mathrm{CRT}(S_2, P_2, s_3, p_3)$
		reconstruct a, b from S_3 (modulo P_3)
		$\mathcal{D}' \leftarrow \{h'_i \in \mathcal{H}' \mid a \bmod h'_i = 0\}$
		if $\prod_{h \in \mathcal{D}'} h \bmod P_3 = a$ then go to final phase
	$\vdots$	
Final phase		
	$\xleftarrow{\mathcal{D}', b}$	
$\mathcal{D} \leftarrow \{h_i \in \mathcal{H} \mid b \bmod h_i\}$		
compute S' from S, $\mathcal{D}$ and $\mathcal{D}'$		
if $\texttt{Hash}(S') \neq H$ go back in phase 1 (event $\perp$)		

Fig. 2. Complete D&F Protocol (for any T).

where κ is the maximum number of rounds needed.

Here are the complexity in the two cases we are interested in:

- if $t_1 = \cdots = t$, $\kappa = \lceil T/t \rceil$, $T_\kappa = \kappa t < T + t$ and the transmission complexity is at most $\frac{5}{2}u(T+t) + \lceil T/t \rceil = O(uT)$;
- if $t_k = 2^k t'$, $\kappa = \lceil \log(T/t') \rceil$, $T_\kappa < 2T$ and the transmission complexity is at most $\frac{5}{2}2uT + \lceil \log(T/t') \rceil = O(ut)$.

We can remark that the first case is slightly better: it may use twice less data. But, as you will see in Section 5.5, the second case is computationally faster.

3.2 Probabilistic Decoding: Reducing p

In this section, we focus on the basic D&F protocol, for the sake of simplicity. However, it can easily be adapted to the complete D&F protocol.

The idea is to generate a prime p about twice shorter than the p recommended in section 2.2, namely:

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

Using notations in Section 2.2, since there are at most t differences, we must have:

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

By opposition to Section 2.2, we do not have a useful fixed bound for a and b anymore; we only have a bound for the *product* ab:

$$ab \leq 2^{ut}. \tag{4}$$

Therefore, we define a sequence of $t+1$ couples of bounds:

$$(A_i, B_i) = \left(2^{ui}, 2^{u(t-i)}\right) \forall i \in \{0, \ldots, t\}$$

Equations (2) and (4) 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, since $A_i B_i = 2^{ut} < p$, given (A_i, B_i, p, s) one can recover (a, b), and hence Oscar can compute $\mathcal{H}'$.

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 (4) for some index $i' \neq i$. We conjecture that such failures happen with negligible probability (that we do not try to estimate here) when u is large enough. In any case, thanks to the global hash $H = \texttt{Hash}(\mathcal{H}')$, if a failure occurs, it can easily be detected. Furthermore, if failures never occur, this variant will roughly halve the quantity of transmitted bits with respect to section 2.2.

4 From Set Reconciliation to File Reconciliation

In this section, we first show how hashing can enable to use D&F or any set reconciliation protocol to do file reconciliation. Then we show how to methods to reduce size of hash, and so transmission cost.

4.1 From Set Reconciliation to File Reconciliation

Let us suppose Oscar and Neil now want to reconcile files, modeled as bit-string of arbitrary lengths, instead of u-bit prime numbers. Let $\mathcal{F} = \{F_1, \ldots, F_n\}$ be the old set of files, owned by Oscar, and let $\mathcal{F}' = \{F'_1, \ldots, F'_n\}$ be the new version of files, owned by Neil. Let η be the total number of files $\eta = |\mathcal{F} \cup \mathcal{F}'| \leq n + n'$.

A naive way to use a set reconciliation for sets of prime numbers (such as D&F) to reconcile $\mathcal{F}$ and $\mathcal{F}'$ is simply to hash each file, or the content of each file, and to reconcile the sets $\mathcal{H}$ and Set' of hashes of the files of Oscar and Neil. Then, when the protocol succeed, Neil can send to Oscar the actual content of the files, corresponding to hashes in $SetD'$, *i.e.*, the files Oscar is missing.

More formally, we have:

$$\begin{array}{lllll} h_1 = \texttt{HashPrime}(F_1) & \ldots & h_n = \texttt{HashPrime}(F_n) & \mathcal{H} = \{h_1, \ldots, h_n\} \\ h'_1 = \texttt{HashPrime}(F'_1) & \ldots & h'_{n'} = \texttt{HashPrime}(F'_{n'}) & \mathcal{H}' = \{h'_1, \ldots, h'_{n'}\}, \end{array}$$

where `HashPrime` is a collision-resistant hash function `HashPrime` to prime numbers, in order for any h_i or h'_i to identify uniquely its corresponding file F_i or F'_i. In Section 4, we show how to construct such hash function from classical collision-resistant hash functions (to bit-strings of fixed length). If we suppose that the output of `HashPrime` is uniform[3], since there are about $\frac{2^m}{m}$ (TODO ref) primes number of size m, we just need to be the size u of the primes output by `HashPrime` to be such that $u - \log u = 2 \log \eta$, because of the birthday paradox (TODO ref). This means that $u \approx 2 \log \eta$.

4.2 The File Laundry: Reducing u

What happens if we brutally shorten the hash size u in the D&F protocol for files?

As expected by the birthday paradox, we should start seeing collisions. In this section, after showing how to deal with these collisions and adapting D&F subsequently, we analyse the statistics governing the appearance of collisions and try to propose some ways to correctly choose u.

[3] This is the case if we use our first proposed Algorithm 1 with a classical hash function which can be modeled as a random function. We suppose this is true for well-known hash functions such as SHA-256. (TODO ?)

New Protocol We remark that a collision can only yield a *false positive*, and never a *false negative*. In other words, while a collision may obliviate a difference[4] a collision will never create a nonexistent difference *ex nihilo*.

Thus, it suffices to replace the hash of files $h_i = \texttt{HashPrime}(F_i)$ (or $h'_i = \texttt{HashPrime}(F'_i)$) by $\hbar_{i,\ell} = \texttt{HashPrime}(\ell|F_i)$ (or $\hbar_{i,\ell} = \texttt{HashPrime}(\ell|F'_i)$ to quickly filter-out file differences by repeating our D&F protocol for $\ell = 1, 2, \ldots$. At each iteration the parties will detect files not in common (*i.e.*, files F_i or F'_i whose hash $\hbar_{i,\ell}$ or $\hbar'_{i,\ell}$ is not colliding), fix these (*i.e.*, remove these from $\mathcal{F}$ and $\mathcal{F}'$) and "launder" again the remaining files $\mathcal{F}$ and $\mathcal{F}'$.

We still need a condition to stop "laundering" files, *i.e.*, a collision to be sure there were no problematic collision. Before showing this condition, let us first analyse precisely which collisions can appear.

Let us consider the execution ℓ of D&F. There are three kinds of collisions:

1. collisions common files of Oscar and Neil, *i.e.*, collisions in $\mathcal{F} \cap \mathcal{F}'$;
2. collisions between a common file of Oscar and Neil, and a file not in common, *i.e.*, collisions between $\mathcal{F} \cap \mathcal{F}'$ and $\mathcal{F} \Delta \mathcal{F}' = (\mathcal{F} \setminus \mathcal{F}') \cup (\mathcal{F}' \setminus \mathcal{F})$
3. collisions between files not in common, *i.e.*, files in $\mathcal{F} \Delta \mathcal{F}'$.

First kind of collisions is obviously not a problem at all. Second kind of collisions can be easily detected by Oscar or Neil, at the end of the execution ℓ of the D&F protocol. This will prevent them from knowing easily to which file F corresponds a hash not in common h. If such collision arrives, the corresponding file will not be reconciled in this execution of D&F and another execution of D&F is required. Third kind of collisions obliviate a real difference between $\mathcal{F}$ and $\mathcal{F}'$. If no mechanism is added, these collisions can never be

Therefore, we propose the following method to check termination. Before anything else is send, Neil needs to send a global hash H' on a collision-resistant hash of files: $H' = \texttt{Hash}((\texttt{Hash}(F'_1), \ldots, \texttt{Hash}(F'_{n'})))$. Contrary to H in $D\&F$, this hash is on a collision-resistant hash of files and not on (potentially colliding) diversified hash $\texttt{HashPrime}(\ell|F)$. Then, if the execution ℓ of the D&F protocol is successful, Neil sends the list of $\texttt{Hash}(F'_i)$ for the new files F'_i, *i.e.* files F'_i in $\mathcal{F}' \setminus \mathcal{F}$ whose hash $\hbar'_{i,\ell}$ is not colliding. Then Oscar can check whether a collision of the third kind appears by using H'.

Naive Analysis We slightly change notations and write:

$$\mathcal{F} \cup \mathcal{F}' = \{F_1, \ldots, F_\eta\}.$$

In this naive analysis, we just compute the number of probability that a file keeps colliding for λ rounds, and we do as if we do not remove files Consider `HashPrime` as a random function from $\{0,1\}^*$ to $\{0, \ldots, 2^u - 1\}$. Let X_i be the random variable:

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

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

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

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

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

Fabrice: already said in Section 2.3... and maybe it is better to say that, in this first analysis, we suppose we do not filter out files, because this only improves the algo...

However, it is important to note that

Assume that the diversified $\hbar_\ell(F)$'s are random and independent. To understand why the probability that a stubborn file persists colliding decreases exponentially with the number of iterations λ, assume that η remains invariant between iterations and define the following random variables:

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

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

By independence, we have:

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

Therefore the average number of colliding files is:

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

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

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

For the previously considered instance[5] we get $\epsilon_2 \leq 5.43\%$ and $\epsilon_3 \leq 2 \cdot 10^{-3}\%$.

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

A more refined (but somewhat technical) analysis. TODO: ACTUALLY NOT CORRECT BECAUSE WE FORGOT COLLISIONS BETWEEN FILES NOT IN COMMON AND FILES IN COMMON !!!!!!!!!!!!!!!!!!!!!!

In Appendix A, we show a more refined analysis, which proves that the survival probability of at least one false positive after k iterations satisfies:

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

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

How to select u? For the sake of simplicity, we consider $t = t_1 = t_2 = \ldots$. For a fixed λ, ϵ'_λ decreases as u grows. For a fixed u, ϵ'_λ also decreases as λ grows. Transmission, however, grows with both u (bigger digests) and k (more iterations). We write for the sake of clarity: $\epsilon'_\lambda = \epsilon'_{\lambda,u,\eta}$.

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:

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

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

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

$\phi_{10^6}(u)$ admits an optimum for $u = 19$.

Note: The previous analysis is incomplete because of the following approximations:

- We consider u-bit prime digests while u-bit strings contain only about $2^u/u$ primes.
- We used a fixed u in all rounds. Nothing forbids using a different u_ℓ at each iteration, or even fine-tuning the u_ℓ's adaptively as a function of the laundry's effect on the progressively reconciliated multisets..
- Our analysis treats 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 *supra*.

A different approach is to optimize t and u experimentally, *e.g.*, using the open source D&F program `btrsync` developed by the authors (*cf.* Section 7).

5 Computational Complexity

In this section, we are interested in computing the computational complexity of our protocol. To simplify the analysis, we assume that there are no collisions, and that $n = n'$.

In this section, after briefly mentioning the cost of a naive implementation, we propose four optimizations to speed up our algorithms. A summary of all costs can be found in Table 1.

5.1 Basic Complexity

Let $\mu(l)$ be the time required to multiply two l-bit numbers[6]. For naive (*i.e.*, convolutive) algorithms $\mu(l) = O(l^2)$, but using FFT multiplication [9], $\mu(l) = \tilde{O}(l)$. FFT is experimentally faster than convolutive methods starting at $l \sim 10^6$. The modular division of two l-bit numbers and the reduction of a $2l$-bit number modulo a l-bit number are also known to cost $\tilde{O}(\mu(l))$ [2]. Indeed, in packages such as `gmp`, division and modular reduction run in $\tilde{O}(l)$, for sufficiently large l.

As proven in Section 5.2, the naive complexity of `HashPrime` is $u^2\mu(u)$, but it can be slightly improved at the expense of making the hashing less uniform. Hence, we have the costs depicted in the third column of Table 1, if we use the same t for each round ($t = t_1 = t_2 = \dots$).

5.2 Hashing Into Primes

Hashing into primes is frequently needed in cryptography. A recommended implementation of $\texttt{HashPrime}(F)$ is given in Algorithm 1. If u is large enough (*e.g.* 160) one might sacrifice uniformity to avoid repeated file hashing by defining $\texttt{HashPrime}(F) = \texttt{NextPrime}(\texttt{Hash}(F))$. Yet another acceleration (that further destroys uniformity) consists in replacing `NextPrime` by Algorithm 2 where $\alpha = 2 \times 3 \times 5 \times \cdots \times \texttt{Prime}[d]$ is the product of the first primes until some rank d.

analyse ci-contre

Fabrice: Cela accélère un peu, car il y a environ $\frac{n}{\log(n)\varphi(\alpha)}$ nombres premiers $\leq n$ et congrus à 1 modulo α, contre $\frac{n}{\log(n)}$ nombres premiers $\leq n$ (voir `http://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_de_la_progression_arithm%C3%A9tique#Version_quantitative`). Dans l'algo 2, h est donc premier avec proba $\frac{\frac{n}{\log(n)\varphi(a)}}{\frac{n}{\alpha}} = \frac{\alpha}{\log(n)\varphi(\alpha)}$, tandis que dans l'algo 1, h est premier avec proba $\frac{1}{\log(n)}$. On a alors une accélération d'environ 10 si on prend $d = 60$.

TODO: no more uniform $\longrightarrow$ slightly decrease entropy of hashing

[6] We assume that $\forall l, l', \mu(l + l') \geq \mu(l) + \mu(l')$.

Algorithm 1 Possible Implementation of $\texttt{HashPrime}(F)$

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

Algorithm 2 Fast Nonuniform Hashing Into Primes

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

5.3 Adapting p_k

Taking the p_k's as ut_k-bit primes is not very practical, because generating a big prime number is slow, and storing a list of such primes require to be able to bound the number of rounds, and to fix all the parameters (u and $t_1, t_2, \dots$). That is why, in this section we show that we can adapt p_k to be able to generate them easily and also, to speed up the computations with a constant factor.

Let $\texttt{Prime}[i]$ denote the i-th prime[7]. Besides conditions on size, the *only* property required from p is to be co-prime with all the h_i's and all the h'_i's. We can hence consider the following variants:

Variant 1: Smooth p_k:

$$p_k = \prod_{j=r_k}^{r_{k+1}-1} \texttt{Prime}[j],$$

where the bounds r_k are chosen to ensure that each p_k has the proper size. Generating such a prime is much faster than generating a big prime p_k.

Variant 2: $p_k = \texttt{Prime}[k]^{r_k}$ where the exponents r_k are chosen to ensure that each p_k has the proper size. This variant is even faster than the previous one, but require to choose h_k bigger than all $\texttt{Prime}[k]^{r_k}$.

Variant 3: $p_k = P_k = 2^{ut_k}$. In this case, $C_k = \prod_{i=1}^{n} h_i \bmod p_k$, $c_1 = C_1$ and $c_k = (C_k - C_{k-1})/p_{k-1}$, *i.e.*, c_k is the slice of bits $ut_{k-1} \dots ut_k - 1$ of C_k (with $T = 0$), which we write $c_k = C_k[ut_{k-1} \dots ut_k]$. Using Algorithm 3 for the computation of c_k, this variant

[7] with $\texttt{Prime}[1] = 2$

offers substantial constant-factor accelerations compared to previous variants, because it removes the need of all modulo operations and of CRT re-combination when multiple p_k are needed, since C_k is just the binary concatenation of c_k and C_{k-1} !

Let us explain the main ideas of Algorithm 3. Let $X_i = \prod_{j=1}^{n} h_j$ ($X_0 = 1$), $X_{i,k} = X_i[ut_{k-1} \dots ut_k]$ and let $D_{i,k}$ be the u upper-bits of the product of $Y_{i,k} = X_i[0 \dots ut_k]$ and h_i, *i.e.*, $D_{i,k} = (Y_{i,k} \times h_i)[ut_k \dots u(t_k+1)]$ ($D_{i,0} = 0$ and $D_{0,k} = 0$). Since $X_{i+1} = X_i \times h_{i+1}$, we have, for $k \geq 0$, $i \geq 0$:

$$\begin{aligned} D_{i+1,k+1} \times 2^{ut_{k+1}} + Y_{i+1,k+1} &= Y_{i,k+1} \times h_{i+1} \\ &= (X_{i,k+1} \times 2^{ut_k} + Y_{i,k}) \times h_{i+1} \\ &= X_{i,k+1} \times 2^{ut_k} \times h_{i+1} + Y_{i,k} \times h_{i+1} \\ &= X_{i,k+1} \times h_{i+1} \times 2^{ut_k} + (D_{i,k} \times 2^{ut_k} + \dots). \end{aligned}$$

Therefore, if we only consider bits $[ut_k \dots u(t_{k+1}+1)]$, for $k \geq 1$,$i \geq 0$:

$$D_{i+1,k+1} \times 2^{u(t_{k+1}} + X_{i+1,k+1} = X_{i,k+1} \times h_{i+1} + D_{i,k}.$$

Since $c_k = X_{n,k}$, the algorithm is correct.

We remark we only need to store $D_{k,i}$ and $D_{k+1,i}$ during round k (for all i). So the space complexity is $O(nu)$.

Algorithm 3 Computation of c_k for $p_k = 2^{ut_k}$

Require: k, the set h_i, $(D_{k,i})$ as explained in the article
Ensure: $c_{k+1} = \prod_{i=1}^{n} h_i \bmod p_{k+1}$, (D_{k+1}) as explained in the article
1: **if** $k = 0$ **then**
2: $\quad X \leftarrow 1$
3: **else**
4: $\quad X \leftarrow 0$
5: **for** $i = 0, \dots, n-1$ **do**
6: $\quad Z \leftarrow X \times h_{i+1}$
7: $\quad D_{i+1,k+1} \leftarrow Z[u(t_{k+1} - t_k) \dots u(t_{k+1} - t_k + 1)]$
8: $\quad X \leftarrow Z[0 \dots u(t_{k+1} - t_k)]$
9: $c_{k+1} \leftarrow X$

5.4 Algorithmic Optimizations using Product Trees

The non-overwhelming (but nonetheless important) complexities of the computations of (c, c') and of the factorizations can be even reduced to $\tilde{O}(\frac{n}{t_k}\mu(ut_k))$ and $\tilde{O}(\frac{n}{T_k}\mu(uT_k))$ using product trees. Therefore, the cost drops to $\tilde{O}(nu)$ with FFT [9]. To simplify the presentation,

we suppose there is only one round, and $p = p_1$, $t = t_1 = T_1$, and we assume that $t = 2^\tau$ is a power of two dividing n. We also only focus on the case p prime, for the sake of simplicity. But the algorithm can be easily adapted to the other variants.

The idea is the following: group h_i's by subsets of t elements and compute the product of each such subset in $\mathbb{N}$:

$$H_j = \prod_{i=(j-1)t+1}^{jt} h_i \in \mathbb{N},$$

for $j \in \{1, \ldots, n/t\}$.

Each H_j can be computed in $\tilde{O}(\mu(ut))$ using the standard product tree method described in Algorithm 4 (for $j = 1$) illustrated in Figure 4. Thus, all these $\frac{n}{t}$ products can be computed in $\tilde{O}(\frac{n}{t}\mu(ut))$. We can then compute c by multiplying the H_j modulo p, which costs $\tilde{O}(\frac{n}{t}\mu(ut))$.

Algorithm 4 Product Tree Algorithm

Require: the set h_i
Ensure: $\pi = \pi_1 = \prod_{i=1}^{t} h_i$, and π_i for $i \in \{1, \ldots, 2t-1\}$ as in Figure 4

```
 1: π ← array of size t
 2: function prodTree(i,start,end)
 3:     if start = end then
 4:         return 1
 5:     else if start + 1 = end then
 6:         return h_{start+1}
 7:     else
 8:         mid ← ⌊(start+end)/2⌋
 9:         π_{2i} ←prodTree(2i,start,mid)
10:         π_{2i+1} ←prodTree(2i + 1,mid,end)
11:         return π_{2i} × π_{2i+1}
12: π_1 ←prodTree(1, 0, t)
```

What is the caveat?

The same technique applies to factorization[8], but with a slight *caveat*.

After computing the tree product, we can compute the residues of a modulo H_1. Then we can compute the residues of a mod H_1 modulo the two children π_2 and π_3 of $H_1 = \pi_1$ in the product tree (depicted in Figure 4), and so on. Intuitively, we descend the product tree doing modulo reduction. At the end (*i.e.*, as we reach the leaves), we obtain the residues of a modulo each of the h_i ($i \in \{1, \ldots, t\}$). This is described in Algorithm 5 and illustrated in Figure 5. We can use the same method for the tree product associated to any H_j, and the residues of a modulo each of the h_i ($i \in \{(j-1)t+1, \ldots, jt\}$) for any j, *i.e.*, a modulo each of the h_i for any i. Complexity is $\tilde{O}(\mu(ut))$ for each j, which amounts to a total complexity of $\tilde{O}(\frac{n}{t}\mu(ut))$.

[8] We explain the process with a, this is applicable *ne variatur* to b as well.

Algorithm 5 Division Using a Product Tree

Require: $a \in \mathbb{N}$, π the product tree of Algorithm 4
Ensure: $A[i] = a \bmod \pi_i$ for $i \in \{1, \ldots, 2t-1\}$, computed as in Figure 5

```
A ← array of size t
function modTree(i)
    if i < 2t then
        A[i] ← A[⌊i/2⌋] mod π_i
        modTree(2i)
        modTree(2i + 1)
A[1] ← a mod π_1
modTree(2)
modTree(3)
```

5.5 Doubling

As seen in Section 2.3, instead of naively using $t_1 = t_2 = \cdots = t$, we can use $t_k = 2^k t'$, and double t_k at each iteration. This increases slightly the transmission complexity, but reduces drastically the computational complexity as you can see in Table 1.

5.6 Summary

Summary of costs is depicted in Table 1.

6 From File Reconciliation to File Synchronization

The previous sections presented how to perform set reconciliation on a set of fixed-size hashes, and how to choose the hash size when it is used to represent the content of files. This suggests a simple protocol to synchronize a set of files: first synchronize the set of hashes as described, then exchange the content of the files corresponding to the hash values that were found to be in the symmetric difference.

However, in practice, we do not wish to synchronize file *sets*, but file *hierarchies*: we are not just interested in the *content* of the files but in their *metadata*. The most important metadata is the file's path (i.e., its name and its location in the filesystem), though other kinds of metadata exist (e.g., modification time, owner, permissions). In many use cases, the file metadata can change without changes in the file content: for instance, files can be *moved* to a different location. When performing reconciliation, we must be aware of this fact, and reflect file moves without re-transferring the contents of the moved files (which is an improvement over popular synchronization tools such as `rsync`).

In this section, we present how to solve this *file synchronization* problem, using our set reconciliation program as a building block.

Entity	Computation	Complexity in $\tilde{O}$ of		Optimization used
		Basic algo.	**Opt. algo.**[a]	
Both	computation of h_i and h'_i	$nu^2\mu(u)$	$\frac{a}{\phi(a)}nu^2\mu(u)$	fast hashing (5.2)
	for round i			
Both	compute redundancies c_i and c'_i	$n \cdot \mu(ut_i)$	$\frac{n}{t_i} \cdot \mu(ut_i)$	product trees (5.4)
Neil	compute $s_i = c'_i/{c_i}$[b] or $S_i = C'_i/{C_i}$[c]	$\mu(uT_i)$	$\mu(uT_i)$	
Neil	compute S_i from S_{i-1} and s_i (CRT)[b]	$\mu(uT_i)$	n/a	use $p_k = 2^{ut_k}$ (5.3)
Neil	find a_i, b_i such that $S_i = a_i/b_i \bmod p_i$	$\mu(uT_i)$	$\mu(uT_i)$	
Neil	factor a_i	$n \cdot \mu(uT_i)$	$\frac{n}{T_i} \cdot \mu(uT_i)$	product trees (5.4)
	last round			
Oscar	factor b_i	$n \cdot \mu(ut_i)$	$\frac{n}{t_i} \cdot \mu(ut_i)$	product trees (5.4)
	overwhelming complexity	TODO	TODO	previous opt. + doubling (5.5)

[a] using all the optimizations of Sections 5.2, 5.3 ($p_k = 2^{ut_k}$), 5.4 and 5.5 (we suppose that the basic algo use $t_1 = t_2 = \cdots = t$. In addition, choosing $p_k = 2^{ut_k}$ provides substantial constant factor accelerations which cannot be seen in this table due to the fact only asymptotic complexity is showed.
[b] only for p_i prime or equivalent TODO;
[c] only for $p_i = 2^{ut_i}$.
[d] using advanced algorithms in [8,12] — naive extended GCD leads $(uT_i)^2$.

Table 1. Global Protocol Complexity

6.1 General Principle

To perform file synchronization, Oscar and Neil will compute the hash of the contents of each of their file (which we will denote as the *content hash*). They will then compute the hash of the concatenation of the content hash and file metadata (which we will call the *mixed hash*). Oscar and Neil then perform set reconciliation on the set of mixed hashes.

Once the reconciliation has completed, all mixed hashes common to Oscar and Neil represent files which have the same content and metadata in Oscar and in Neil. Neil now sends to Oscar the content hashes and metadata of the files whose mixed hash appears in Neil but not in Oscar. Oscar is now aware of the metadata and content hash of all of Neil's files that do not exist in Oscar with the same content and metadata (we will call them the *missing files*).

Oscar now looks at the list of content hashes of the missing files. For some of these hashes, Oscar may already have a file with the same content hash, only with the wrong metadata. For others, Oscar may not have any file with the same content hash. In the first case, Oscar can recreate Neil's file by altering the metadata, without retransferring the file contents. In the second case, Oscar needs to retrieve the full file contents from Neil. To complete the file synchronization, we first deal with all of Oscar's files which fall in the first case: this is a bit tricky because "altering the metadata" is not obvious to perform in-place when it involves file moves, and is the focus of Section 6.2. Once this has been performed, we transfer all missing files: this is explained in Section 6.3.

6.2 Moving Existing Files

To reproduce the structure of Oscar on Neil's disk, we need to perform a sequence of file moves. 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, but b already exists), or because a folder cannot be created (we want to move a to b/c, but b already exists as a file and not as 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 absent on Oscar, then we can just delete the blocker. However, if a blocker 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 whose 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 the operations 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 could lead to problems if the program is interrupted. We can do something more clever by performing a decomposition in Strongly Connected Components (SCC) 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 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 6.

TODO: $\mathfrak{D}$: collision of notation with $\mathcal{D}$.

6.3 Transferring Missing Files

Once all moves have been applied, Oscar's hierarchy contains all of its files which also exist on Neil, and they have been put at the correct location. The only thing that remains is to transfer the contents of Neil's files that do not exist in Oscar's hierarchy and create those files at the right position. To do so, a simple solution is to use `rsync` to synchronize explicitly the correct files on Neil to the same location in Oscar's hierarchy, using the fact that Oscar is now aware of all of Neil's files and their location.

Algorithm 6 Perform Moves

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

```
1: M ← []
2: T ← []
3: for f in 𝔇's keys do
4:     M[f] ← 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:             delete(b)                          ▷ We don't need b
9:         else
10:            RESOLVE(b)                         ▷ Take care of b and make it go away
11:    if T[f] was set then
12:        f ← T[f]
13:    copy(f, d)
14: function RESOLVE(f)
15:    if M[f] = done then
16:        return                                 ▷ Already managed by another in-edge
17:    if M[f] = doing then
18:        T[f] ← mktemp()
19:        move(f, T[f])
20:        M[f] ← done
21:        return                                 ▷ We found a loop, moved f out of the way
22:    M[f] ← doing
23:    for d ∈ 𝔇[f] do
24:        if d ≠ f then
25:            UNBLOCK_COPY(f, d)                 ▷ Perform all the moves
26:    if f ∉ 𝔇[f] and T[f] was not set then
27:        delete(f)
28:    if T[f] was set then
29:        delete(T[f])
30: for f in 𝔇's keys do
31:    RESOLVE(f)
```

7 Implementation

We implemented the D&F set reconciliation protocol, extended it to perform file synchronization, and benchmarked it against `rsync`. The implementation is called `btrsync`, its source code is available from [1]; it was written using the Python programming language (using `gmp` to perform the number theoretic operations), and using a bash script (invoking ssh) to create the communication channel between the two hosts.

7.1 Implementation Choices

Our implementation does not take into account all the possible optimizations described in Section 5: it implements doubling (Section 5.5) and uses powers of small primes for the p_k (Variant 2 of Section 5.3), but does not implement product trees (Section 5.4) or use the prime hashing scheme described in Section 5.2.

As for file synchronization (Section 6), The only metadata managed by `btrsync` is the file path (name and location). Other kinds of metadata (modification date, owner, permissions) are not implemented, though it would be easy to do so. An optimization implemented by `btrsync` over the move resolution algorithm described in Section 6.2 is to avoid doing a copy of a file F and then removing F: the implementation replaces such operations by move operations, which are faster than copies on most filesystems as the OS does not need to copy the actual file contents.

7.2 Experimental Comparison to `rsync`

We compared `rsync`[9] and our implementation `btrsync`. The directories used for the benchmark are described in Table 2, and the options used when benchmarking `rsync` are given in Table 3. Experiments were performed without any network transfer, by synchronizing two folders on the same host. Hence, time measurements mostly represent the CPU cost of the synchronization.

Results are given in Table 4. In general, `btrsync` spent more time than `rsync` on computation (especially when the number of files is large, which is typically seen in the experiments involving `synthetic`). Transmission results, however, turn out to be favorable to `btrsync`.

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

On non-trivial tasks, however, `btrsync` outperforms `rsync`. This is the case of the `synthetic` datasets, where `btrsync` does not have to transfer information about all unmodified files, and even more so in the case where there are no modifications at all. For

[9] `rsync` version 3.0.9, used both as a competitor to benchmark against and as an underlying call in our own code.

Firefox source code datasets, `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 avoid retransferring the moved folder.

8 Conclusion and Further Improvements

TODO write a conclusion.

Be more specific!

Many fine questions of the probabilistic discussions in the paper are left as future work. Another further line of research would be to pursue development of `btrsync` to make it suitable for end users.

Another possible line of future work is to try to detect files that have been moved and altered slightly. With the current algorithm, the file hash will be different and the whole file will be retransferred, unless if the file location hasn't changed and the underlying call to `rsync` can perform in a clever manner.

8.1 Future work: Remove Hashing to Prime Numbers

In most cases, the most costly operation is the hashing to prime number. That is why, a further optimization can consist in removing the need to hash to prime numbers by hashing to any integer coprime with all the p_i. The problem is that, even if there are no collision and a and b are correctly recovered, the fact that h_i divides a does not mean F_i should be a file in $\mathcal{S}$, because, for example, we can have $a = 150$, $h_1 = 10$, $h_2 = 15$, $h_3 = 6$, and $a = h_1 \times h_2$, but h_3 divides a. Therefore, we need a slightly more complex method for the factorization and a careful probability analysis to ensure this case does not come too often. we should implement the non-prime version stuff... but difficult to find the correct subset of hi which factorizes... This is left for future work.

Other future work: Pour éviter le cas empty $\rightarrow$ source trop gros, on pourrait imaginer l'astuce suivante: si jamais Neil la taille de c est plus petite que la taille du produit des nombres premiers $p_1...p_n$ utilisés, Neil envoie un message pour l'indiquer, et on arrête là le protocole. Et Oscar peut directement factoriser ce nombre envoyé...

9 Acknowledgment

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

10 ToDo

- Refaire une dernière fois les expériences, vu que Fabrice a significativement amélioré les perfs.
- Fabrice: comparer nos perfs avec http://ipsit.bu.edu/programs/reconcile/ ? ou pas ?
- S'assurer qu'on explique explicitement comment/pourquoi on passe de set reconciliation (sync bidirectionnelle) à neil et oscar (sync unidirectionnelle).

References

1. Amarilli, A., Ben Hamouda, F., Bourse, F., Morisset, R., Naccache, D., Rauzy, P.: `https://github.com/RobinMorisset/Btrsync`
2. Burnikel, C., Ziegler, J., Stadtwald, I., D-Saarbrucken: Fast recursive division (1998)
3. Eppstein, D., Goodrich, M., Uyeda, F., Varghese, G.: What's the difference?: efficient set reconciliation without prior context. In: ACM SIGCOMM Computer Communication Review. vol. 41, pp. 218–229. ACM (2011)
4. 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)
5. Hofmann, M., Pierce, B.C., Wagner, D.: Edit lenses. In: Field, J., Hicks, M. (eds.) POPL. pp. 495–508. ACM (2012)
6. Minsky, Y., Trachtenberg, A.: Scalable set reconciliation. In: 40th Annual Allerton Conference on Communications, Control and Computing (October 2002), a full version can be downloaded from `http://ipsit.bu.edu/documents/BUTR2002-01.ps`
7. Minsky, Y., Trachtenberg, A., Zippel, R.: Set reconciliation with nearly optimal communication complexity. Information Theory, IEEE Transactions on 49(9), 2213–2218 (2003)
8. Pan, V., Wang, X.: On rational number reconstruction and approximation. SIAM Journal on Computing 33, 502 (2004)
9. Schönhage, A., Strassen, V.: Schnelle multiplikation grosser zahlen. Computing 7(3), 281–292 (1971)
10. Tridgell, A.: Efficient algorithms for sorting and synchronization. Ph.D. thesis, PhD thesis, The Australian National University (1999)
11. Vallée, B.: Gauss' algorithm revisited. J. Algorithms 12(4), 556–572 (1991)
12. Wang, X., Pan, V.: Acceleration of Euclidean algorithm and rational number reconstruction. SIAM Journal on Computing 32(2), 548 (2003)

A Refined Analysis of the File Laundry

TODO: NOT CORRECT BECAUSE WE FORGOT COLLISIONS BETWEEN FILES NOT IN COMMON AND FILES IN COMMONS.

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 file-pairs (F_i, F_j) such that F_i and F_j have both already collided during *all the previous iterations*[10]. TODO this sentence is unclear: We call such collisions "masquerade balls" (*cf.* Figure 3). Define the two random variables:

[10] Note that we do not require that F_i and F_j repeatedly collide *which each other*; *e.g.*, we may witness during the first round $\hbar_1(F_1) = \hbar_1(F_2)$, $\hbar_1(F_3) = \hbar_1(F_4)$ and $\hbar_1(F_5) = \hbar_1(F_6)$ while during the second round $\hbar_2(F_1) = \hbar_2(F_2)$, $\hbar_2(F_3) = \hbar_1(F_6)$ and $\hbar_2(F_5) = \hbar_2(F_4)$ as shown in Figure 3.

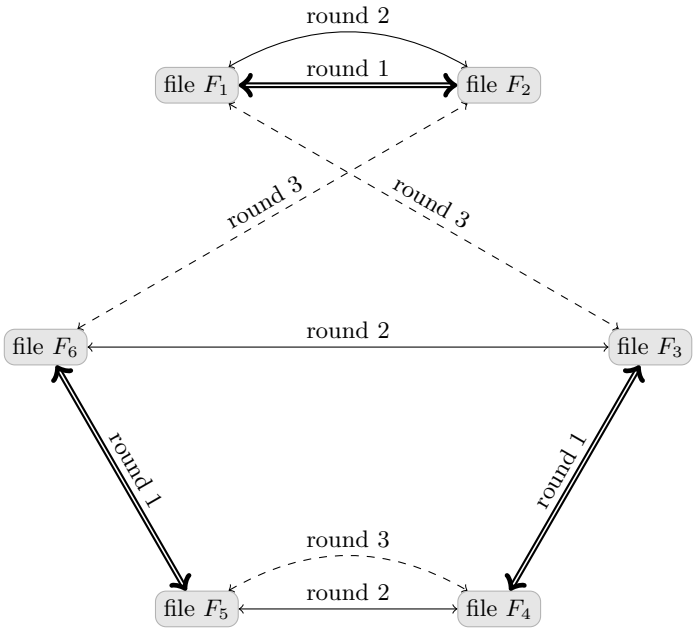

Fig. 3. Illustration of three masquerade balls. Each protocol round is materialized by a different type of arrow. Arrows denotes collisions.

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

$$X_{i,j}^\ell = \begin{cases} 1 & \text{if files } F_i \text{ and } F_j \text{ collide during iteration } \ell. \\ 0 & \text{otherwise.} \end{cases}$$

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

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

Furthermore $p_0 = 1$ and

Fabrice: maybe put this in appendix and add a few comments...

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

hence:

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

and

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

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

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

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

B Figures for Product Tree Optimization

Figures 4 and 5.

C Implementation Details and Benchmarks

TODO: utiliser des préfixes SI pour une meilleure lisibilité

Tables 2, 3 and 4.

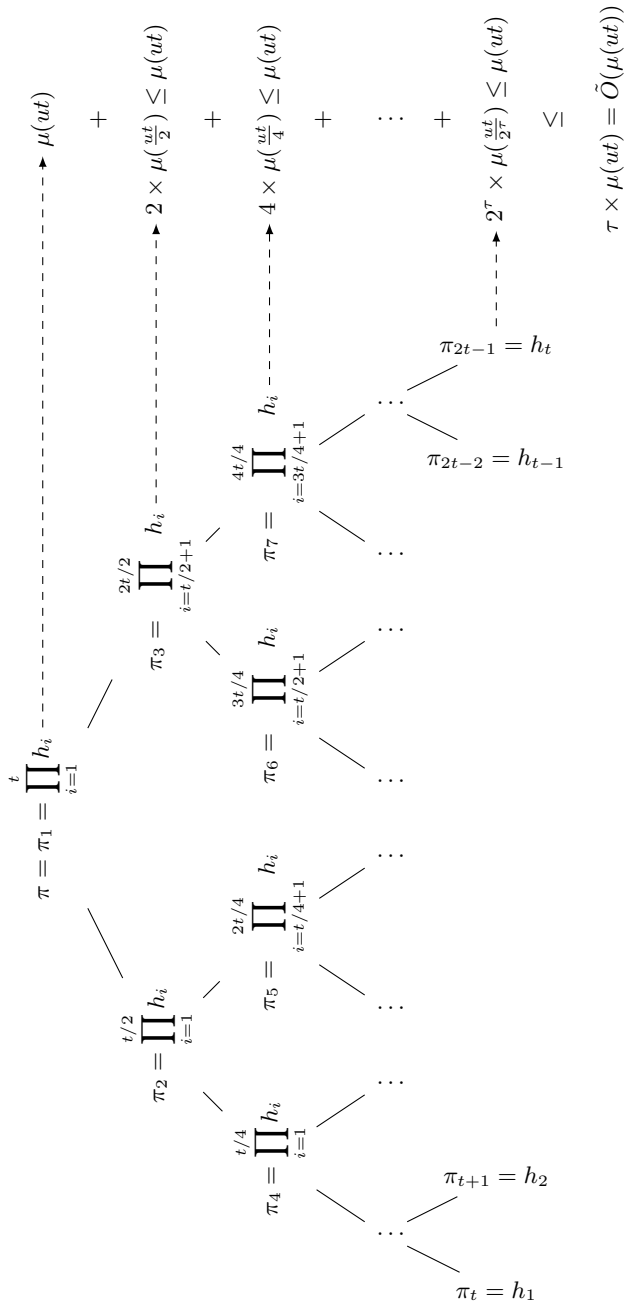

Fig. 4. Product Tree

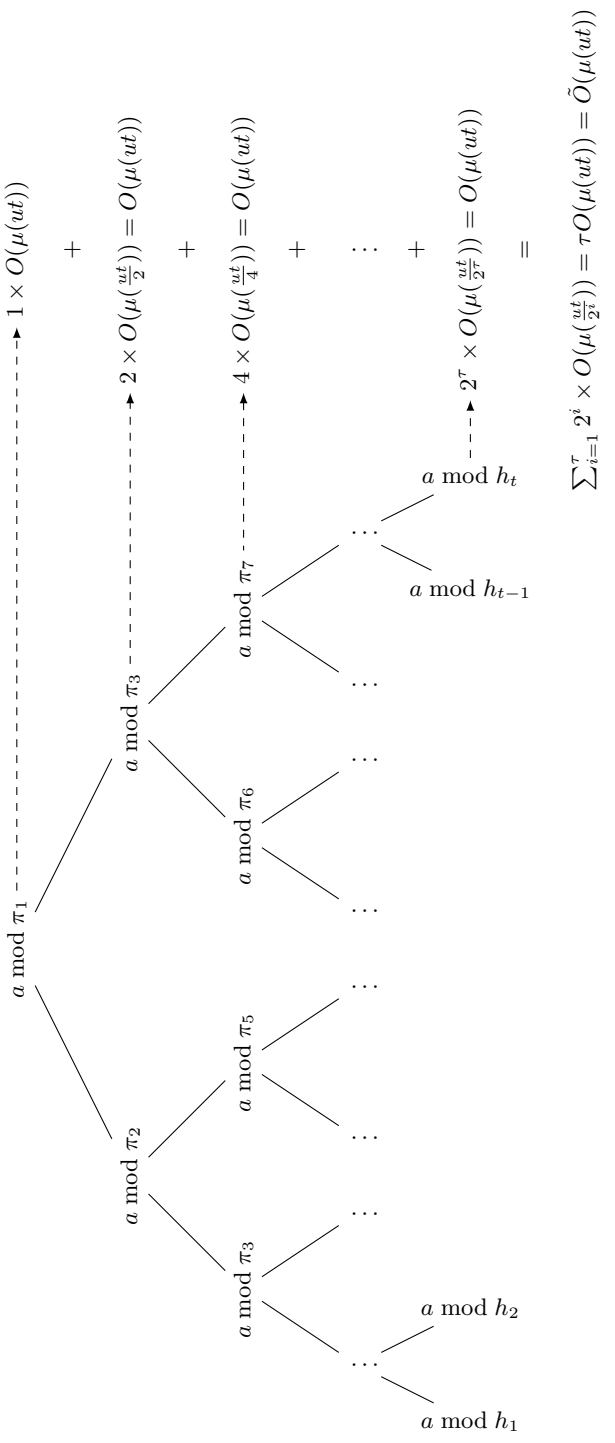

Fig. 5. Modular Reduction From Product Tree

Directory	Description
`synthetic`	A directory containing 1000 very small files containing the numbers $1, 2, \ldots, 1000$.
`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.

Table 2. Test Directories.

▶ `--delete`	Delete existing files on Oscar which do not exist on Neil like `btrsync` does.
▶ `-I`	Ensure that `rsync` does not cheat by looking at file modification times (which `btrsync` does not do).
▶ `--chmod="a=rx,u+w"`	Attempt to disable the transfer of file permissions (which `btrsync` does not transfer). Although these settings ensure that `rsync` does not need to transfer permissions, verbose logging suggests that it does transfer them anyway, so `rsync` must lose a few bytes per file as compared to `btrsync` for this reason.
▶ `-v`	Count the number of sent and received bytes. For `btrsync` we added a piece of code counting the amount of data transmitted during `btrsync`'s own negotiations.

Table 3. Options used when benchmarking `rsync`

Entities and Datasets		Transmission (Bytes)						Time (s)	
Neil's $\mathfrak{F}'$	Oscar's $\mathfrak{F}$	$\mathtt{TX}_{\mathtt{rs}}$	$\mathtt{RX}_{\mathtt{rs}}$	$\mathtt{TX}_{\mathtt{bt}}$	$\mathtt{RX}_{\mathtt{bt}}$	$\delta_{\mathtt{rs}} - \delta_{\mathtt{bt}}$	$\frac{\delta_{\mathtt{bt}}}{\delta_{\mathtt{rs}}}$	$\mathtt{time}_{\mathtt{rs}}$	$\mathtt{time}_{\mathtt{bt}}$
`source`	`empty`	778311	1614	779517	10140	9732	1.0	0.1	0.4
`empty`	`source`	24	12	11927	5952	17843	496.6	0.1	0.4
`empty`	`empty`	24	12	19	30	13	1.4	0.0	0.3
`synthetic`	`synthetic_shuffled`	54799	19012	7308	3417	-63086	0.1	0.2	1.5
`synthetic_shuffled`	`synthetic`	54407	18822	6822	3042	-63365	0.1	0.2	0.8
`synthetic`	`synthetic`	54799	19012	327	30	-73454	0.0	0.1	0.7
`firefox-13.0.1`	`firefox-13.0`	40998350	1187	39604079	3305	-1392153	1.0	1.5	10.2
`source_moved`	`source`	778176	1473	2757	1966	-774926	0.0	0.1	0.6

Table 4. Experimental results. `rs` and `bt` subscripts respectively denote `rsync` and `btrsync`. The two first columns indicate the datasets. Synchronization is performed *from* Neil *to* Oscar. `RX` and `TX` denote the quantity of received and sent bytes and $\delta_{\square} = \mathtt{TX}_{\square} + \mathtt{RX}_{\square}$. $\delta_{\mathtt{rs}} - \delta_{\mathtt{bt}}$ and $\delta_{\mathtt{bt}}/\delta_{\mathtt{rs}}$ express the absolute and the relative differences in transmission between the two programs. The last two columns show timing results. This evaluation was performed on an HP ProBook 5330m laptop using an Intel Core i3-2310M CPU clocked at 2.10 Ghz.