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`)

Si cet abstract vous convient, il faudrait en reprendre des bouts dans l'introduction.

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 number theoretic reconciliation protocol called "Divide & Factor" (D&F) 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` uses more CPU time but transmits less data.

1 Introduction

This work revisits *set reconciliation*, a problem consisting in synchronizing two multisets while minimizing the amount of data transmitted. Set reconciliation arises in many practical situations, the most typical of which is certainly incremental backups performed over a slow network link.

Est-ce que c'est vrai qu'on apporte quelque chose de nouveau dans le cas où c'est des fichiers de taille non fixe ?

Several efficient and elegant solutions are known to achieve set reconciliation of multisets containing atomic elements of a fixed size. For instance, [3] manages to perform set reconciliation using a bandwidth which is linear in the size of the symmetric difference of the multisets multiplied by the size of the elements, which is optimal in this setting. We refer the reader to [3,4,5] (to quote a few references) for more on this problem's history and its existing solutions.

However, in the case where the elements to be synchronized can be very large (e.g., files during a backup), we must use checksums to identify the differing files before transferring them, and the question of the size of the checksum to use is non-trivial. In this article, we propose a new reconciliation protocol called "Divide & Factor" (D&F) based on number theory. In terms of asymptotic transmission complexity, the proposed procedure reaches optimality as well. In addition, the new protocols offer a very interesting gamut of parameter trade-offs. We provide an analysis of the protocol's complexity in terms of transmission and computation, as well as a probabilistic analysis of the possible choices of checksum sizes; we also provide an implementation of the protocol and experimental results.

This paper is structured as follows: Section 2.2 presents a basic version of the proposed protocol. This basic version suffers from two limitations: it works only if the number of differences to reconcile is bound and it may fail leave the synchronized party in an erroneous state. Failure avoidance is overcome in section 2.3 and an extension to arbitrary numbers of differences is given in section 2.4. The protocol's transmission complexity is treated in section 3. Section 3 also introduces two transmission optimizations and analyzes them in detail. Section 4 analyzes the computational complexities of the proposed protocols and 5 reports practical experiments and benchmarks against the popular software `rsync`.

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 new, up-to-date version $\mathfrak{D}'$: $\mathfrak{D}$ and $\mathfrak{D}'$ can differ both in their files and in their tree structures. Oscar wishes to obtain $\mathfrak{D}'$ but *exchange as little data as possible* during the synchronization process.

To tackle this problem we separate the *"what"* from the *"where"* by considering files as a tuple of their location and content. In other words, we will first synchronize all the file contents and then move files to the adequate location. We consider that $\mathfrak{D}$ is a multiset of files which we denote as $\mathfrak{F} = \{F_0, \ldots, F_n\}$, and likewise represent $\mathfrak{D}'$ as $\mathfrak{F}' = \{F'_0, \ldots, F'_{n'}\}$.

Let t_0 be the number of discrepancies between $\mathfrak{F}$ and $\mathfrak{F}'$ that Oscar wishes to learn, *i.e.* the symmetric difference of $\mathfrak{F}$ and $\mathfrak{F}'$:

$$t_0 = \#\mathfrak{F} + \#\mathfrak{F}' - 2\#\left(\mathfrak{F} \bigcap \mathfrak{F}'\right) = \#\left(\mathfrak{F} \bigcup \mathfrak{F}'\right) - \#\left(\mathfrak{F} \bigcap \mathfrak{F}'\right)$$

Given a file F, we denote by $\texttt{Hash}(F)$ its image by a collision-resistant hash function such as SHA-1. Let $\texttt{HashPrime}(F)$[1] be a function hashing files (uniformly) into primes smaller than 2^u for some $u \in \mathbb{N}$. Define the shorthand notations: $h_i = \texttt{HashPrime}(F_i)$ and $h'_i = \texttt{HashPrime}(F'_i)$.

2.2 Description of the Basic Exchanges

The number of differences t_0 is unknown to Oscar and Neil. However, for the time being, we will assume that t_0 is smaller than some t and attempt to perform synchronization. If $t_0 \leq t$, synchronization will succeed; if $t_0 < t$ the parties will transmit more information later to complete the synchronization, as explained in section 2.4.

We generate a prime p such that:

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

[1] The design of `HashPrime` is addressed in Appendix C.

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

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

Neil computes:

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

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

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

Note that if our assumption $t_0 \leq t$ is correct, $\mathfrak{F}$ and $\mathfrak{F}'$ differ by at most t elements and a and b are strictly less than 2^{ut}. The problem of recovering a and b from s efficiently is known as *Rational Number Reconstruction* [4,8]; theorem 1 (see [2]) guarantees that it can be solved in this setting.

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.*

Taking $A = B = 2^{ut} - 1$, Equation (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: a possible option is o use Gauss' algorithm for finding the shortest vector in a bi-dimensional lattice [7]. 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 $\mathfrak{F}$ and $\mathfrak{F}'$ and settle them.

The formal description of the protocol is given in Figure 1. The "output $\perp_{\mathsf{bandwidth},\square}$ " protocol interruptions will:

- never occur if the assumption $t_0 \leq t$ holds.
- occur with high probability if $t_0 > t$. Indeed, for a potential $\perp_{\mathsf{bandwidth},1}$ to be overlooked, the ut-bit number a must perfectly factor over a set of n primes of size u. If we assume that a is "random", the probability γ that a is divisible by some h_i is essentially $\gamma \sim 1/h_i \sim 2^{-u}$, the probability that a is divisible by exactly t digests is:

$$\alpha = \binom{n}{t} \gamma^t (1-\gamma)^{n-t} \sim \binom{n}{t} 2^{-ut} (1 - 2^{-u})^{n-t}$$

  and the probability that the protocol does not terminate by a $\perp_{\mathsf{bandwidth},\square}$ when $t_0 > t$ is $\sim \alpha^2$.

Oscar		**Neil**
compute c		
	$\xrightarrow{c}$	
		compute a, b
		if a doesn't factor as a product of h'_is
		then output $\perp_{\mathsf{bandwidth},1}$ and halt
		$\mathfrak{S} \leftarrow \{F'_i \text{ s.t. } a \bmod h'_i = 0\}$
	$\xleftarrow{\mathfrak{S},b}$	
if b doesn't factor as a product of h_i's		
then output $\perp_{\mathsf{bandwidth},2}$ and halt		
delete files s.t. $b \bmod h_i = 0$		
add $\mathfrak{S}$ to the disk		

Fig. 1. Basic Protocol.

The very existence of $\perp_{\mathsf{bandwidth},\square}$'s is annoying for two reasons:

- A file synchronization procedure that works *only* for a limited number of differences is not really useful in practice. Thus, section 2.4 explains how to extend the protocol to perform the synchronization even when the number of differences t_0 exceeds the initial estimation t.
- If, by sheer bad luck, both $\perp_{\mathsf{bandwidth},\square}$'s went undetected (double accidental factorization) the Basic Protocol (Fig. 1) may leave Oscar in an inconsistent state.

Double accidental factorization is not only possible source of inconsistent states: as we did not specifically require `HashPrime` to be collision-resistant, the events

$$\perp_{\mathsf{collision},1} = \begin{cases} h'_i = h'_j \text{ for } i \neq j \\ \\ a \bmod h_i = 0 \end{cases} \quad \text{and/or} \quad \perp_{\mathsf{collision},2} = \begin{cases} h_i = h_j \text{ for } i \neq j \\ \\ b \bmod h'_i = 0 \end{cases}$$

will cause Neil to send wrong files in $\mathfrak{S}$ ($\perp_{\mathsf{collision},1}$) and/or have Oscar unduely delete files owned by Neil ($\perp_{\mathsf{collision},2}$).

Inconsistent states may hence stem from three events:

- accidental double factorization of a and/or b when $t_0 > t$ (probability α^2)
- $\perp_{\mathsf{collision},1}$ = collisions within the set $\{h'_i\}$
- $\perp_{\mathsf{collision},2}$ = collisions within the set $\{h_i\}$

Section 2.3 explains how protect the protocol from all inconsistent events at once.

2.3 Avoiding Inconsistency

The Basic Protocol of Figure 1 is fully deterministic. Hence if any sort of trouble occurs, repeating the protocol will be of no help. We modify the protocol as follows:

- Let $H \leftarrow \texttt{Hash}(\mathfrak{F}')$
- Replace $\texttt{HashPrime}(F)$ by a diversified $\hbar_k(F) = \texttt{HashPrime}(k|F)$.
- Define the shorthand notations: $\hbar_{k,i} = \hbar_k(F_i)$ and $\hbar'_{k,i} = \hbar_k(F'_i)$.
- Let $\mathsf{StepProtocol}(k)$ denote the sub-protocol shown in Figure 2.
- Use the protocol of Figure 3 as a fully functional reconciliation protocol for $t_0 \leq t$.

Oscar		**Neil**
		if a doesn't factor as a product of $\hbar'_{k,i}$s then output $\perp_{\mathsf{bandwidth},1}$ and halt $\mathfrak{S} \leftarrow \{F'_i \text{ s.t. } a \bmod \hbar'_{k,i} = 0\}$ if there are collisions in $\mathfrak{S}$ then output $\perp_{\mathsf{collision},1}$ and halt
	$\xleftarrow{\mathfrak{S},b}$	
if b doesn't factor as a product of $\hbar_{k,i}$'s then output $\perp_{\mathsf{bandwidth},2}$ and halt $\mathfrak{A} \leftarrow \{F_i \text{ s.t. } b \bmod \hbar_{k,i} = 0\}$ if there are collisions in $\mathfrak{A}$ then output $\perp_{\mathsf{collision},2}$ and halt if $H \neq \texttt{Hash}(\mathfrak{F} \bigcup \mathfrak{S} - \mathfrak{A})$ then output $\perp_{\mathsf{bandwidth},3}$ and halt add $\mathfrak{S}$ to the disk and erase $\mathfrak{A}$ from the disk return $\mathsf{success}$		

Fig. 2. $\mathsf{StepProtocol}(k)$.

Note: To avoid transmitting the (potentially very voluminous) $\mathfrak{S}$ during $\mathsf{StepProtocol}$ before knowing if one of the errors $\perp_{\mathsf{bandwidth},2}, \perp_{\mathsf{bandwidth},3}, \perp_{\mathsf{collision},2}$ will occur, Neil may transmit

$$\mathfrak{S}' = \{\texttt{Hash}(F'_i),\ F'_i \in \mathfrak{S}\}$$

instead of $\mathfrak{S}$ and send $\mathfrak{S}$ only after successfully passing the $\perp_{\mathsf{bandwidth},3}$ test. The definition of H must be changed accordingly to

$$H = \texttt{Hash}(\{\texttt{Hash}(F'_i),\ F'_i \in \mathfrak{F}'\})$$

Oscar		**Neil**
		compute $H \leftarrow \texttt{Hash}(\mathfrak{F}')$
	$\xleftarrow{H}$	
compute c		
	$\xrightarrow{c}$	
		compute a, b
		$k \leftarrow 1$
	$\xleftarrow{\mathsf{StepProtocol}(k)}$	while $\mathsf{StepProtocol}(k) = \bot_{\mathsf{collision},\Box}$
		$\quad k \leftarrow k+1$

Fig. 3. Fully Functional Protocol for $t_0 \leq t$.

2.4 Handling a High Number of Differences

To extend the protocol to an arbitrary t_0, Oscar and Neil agree on an infinite set of primes $p_1, p_2, \ldots$ As long as the protocol fails with a $\bot_{\mathsf{bandwidth},\Box}$ status, 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 $\mathfrak{F}'$.

All $\bot$ treatments were removed from Appendix A for the sake of clarity (these can be very easily added by modifying Appendix A *mutatis mutandis*). In essence, the rules are: add information modulo a new p_i whenever the protocol fails with a $\bot_{\mathsf{bandwidth},\Box}$ and increment k whenever the protocol fails with a $\bot_{\mathsf{collision},\Box}$.

A typical execution sequence is thus expected to be something like:

$$\bot_{\mathsf{bandwidth},1}, \bot_{\mathsf{bandwidth},1}, \bot_{\mathsf{bandwidth},1}, \bot_{\mathsf{bandwidth},1}, \bot_{\mathsf{collision},1}, \bot_{\mathsf{collision},1}, \mathsf{success}$$

3 Transmission Complexity

This section explores two strategies for reducing the size of p and hence improving transmission by *constant factors* (from an asymptotic communication standpoint, improvements cannot be expected as the protocol already transmits information proportional to t_0, the difference to settle). Excluding the core information $\mathfrak{S}$ and assuming that no $\bot_{\mathsf{collision},\Box}$ events occurred, the transmission complexity of the protocol of Appendix A is:

$$\lambda \log(\max_{i=1}^{\lambda} c_i) + \log b \leq \lambda \log(\max_{i=1}^{\lambda} p_i) + \frac{1}{2} \log \prod_{i=1}^{\lambda} p_i \leq 3\lambda(ut_0 + 1) = O(\lambda u t_0) = O(ut)$$

Where $\lambda = t/t_0$ is the number of rounds required to complete the protocol. As we have no control over t, decreasing u is the main natural optimization option. We will get back to this later on in this paper (section 3.2).

3.1 Probabilistic Decoding: Reducing p

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

$$2^{u(t+1)} < p \leq 2^{u(t+1)+1} \tag{2}$$

Let $\eta = \max(n, n')$. The new redundancy c is calculated as previously and is hence also approximately twice smaller. Namely:

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

and 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 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 $t+1$ couples of bounds:

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

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 conjecture that such failures happen with negligible probability (that we do not try to estimate here) when u is large enough, but this makes the modified protocol heuristic only. If failures never occur, this variant will roughly halve the quantity of transmitted bits with respect to section 2.2.

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

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

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 obliviate a difference[2] a collision will never create a nonexistent difference *ex nihilo*.

Thus, it suffices to replace $\texttt{HashPrime}(F)$ by a diversified $\hbar_k(F) = \texttt{HashPrime}(k|F)$ 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 the diversified $\hbar_k(F)$'s are random and independent. To understand why the probability that a stubborn file persists colliding decreases exponentially with the number of iterations k, assume that η remains invariant between iterations and define the following random variables:

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

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

Fabrice je ne vois pas la difference entre X_i^ℓ dans cette section et Z_i^ℓ dans la suivante. Peux-tu preciser STP

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

By independence, we have:

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

Therefore the average number of colliding files is:

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

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

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

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

A more refined (but somewhat technical) analysis. As mentioned previously, the parties can remove the files 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*[4]. We call such collisions "masquerade balls" (*cf.* Figure 4). Define the two random variables:

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

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

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

[4] 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 4.

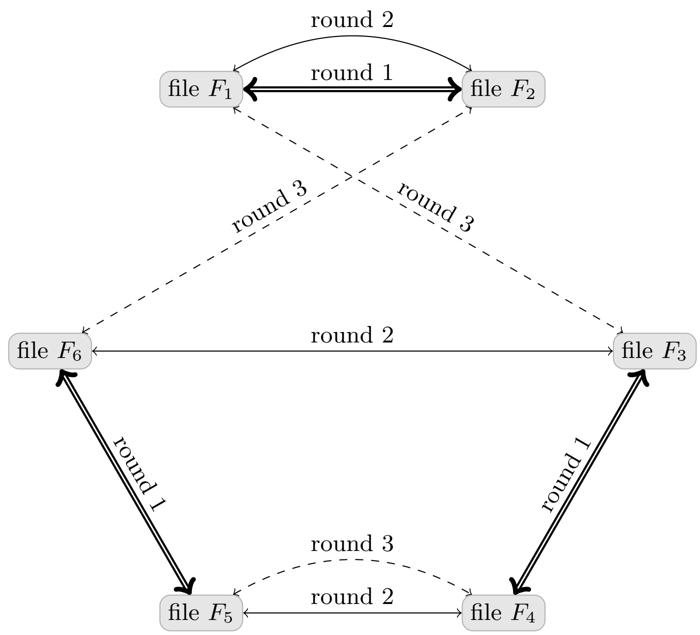

Fig. 4. Illustration of three masquerade balls. Each protocol round is materialized by a different type of arrow. Arrows denotes collisions.

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:

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

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 $\epsilon'_2 \leq 0.013\%$.

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:

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

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

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

$\phi_{10^6}(u)$ 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_k at each iteration, or even fine-tuning the u_ks 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 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 half-sized p. By opposition to section 2.2, confirming correct protocol termination is now non-trivial.

La définition de $\zeta(u)$ est assez gratuite vu que ça n'intervient pas dans la suite.

We say that a *round failure* occurs whenever a round results in an $(a', b') \neq (a, b)$ satisfying Equation (3). Let the round failure probability be some function $\zeta(u)$ (that we did not estimate). If u is kept small (for efficiency reasons), the probability $(1 - \zeta(u))^\ell$ 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* [2]:

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

We refer the reader to [2] for more on this problem, which is beyond our scope. Boneh [1] provides a polynomial-time algorithm for solving the CRWEP under certain conditions satisfied in our setting.

But how can we confirm the solution? As mentioned in section 2.3, Neil will send to Oscar $H = \mathtt{Hash}(\mathfrak{F}')$ as the interaction starts. As long as Oscar's CRWEP resolution will not yield a state matching H, the parties will continue the interaction.

4 Computational Complexity

Let $\mu(k)$ be the time required to multiply two k-bit numbers[5]. For naïve (*i.e.* convolutive) algorithms $\mu(k) = O(k^2)$, but using FFT multiplication [5], $\mu(k) = \tilde{O}(k)$. FFT is experimentally faster than convolutive methods starting at $k \sim 10^6$. The modular division of two

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

k-bit numbers and the reduction of $2k$-bit number modulo a k-bit number are also known to cost $\tilde{O}(\mu(k))$ [1]. Indeed, in packages such as gmp, division and modular reduction run in $\tilde{O}(k)$.for sufficiently large k.

Given that $p \sim 2^{ut}$, we obtain the following complexity analysis:

Je ne comprends pas quelle la différence entre les colonnes 3 4 et les colonnes 5 6 de ce tableau.

Entity	Computation	Complexity expressed in $\tilde{O}$ of			
Both	generate primes	nu^4	Rabin-Miller	nu^3	Rabin-Miller
Both	compute redundancies c and c'	$n \cdot \mu(ut)$	naïve product	nut	using FFT
Oscar	compute $s = c'/c \bmod p$	$\mu(ut)$	naïve inversion	ut	using FFT
Oscar	find a, b such that $s = a/b \bmod p$	$(ut)^2$	naïve ext. GCD	ut	using [4,8]
Both	factor a (resp. b) by modular reductions	$n \cdot \mu(ut)$	naïve reduction	nut	using FFT
	Overwhelming complexity:	$\max(nu^4, n \cdot \mu(ut), (ut)^2)$		$nu \max(t, u^3)$	

Table 1. Global Protocol Complexity. In practice $\max((ut)^2, n \cdot \mu(ut)) = n(ut)^2$ because $(ut)^2 \ll n \cdot \mu(ut) = n(ut)^2$. This boils down to $\max(n(ut)^2, nu^4) = n(u \max(t, u))^2$.

4.1 Improvements Using Product Trees

commIndiquer clairement à quelles lignes du tableau ça se réfère, et si la complexité donnée dans le tableau prend ces optims en compte. The non-overwhelming (but nonetheless important) complexities of the computations of (c, c') and of the factorizations can be even reduced to $\tilde{O}(\frac{n}{t}\mu(ut))$ using convolutive methods. These complexities can be reduced to $\tilde{O}(nu)$ with FFT [5]. To simplify the presentation, assume that $t = 2^\tau$ is a power of two dividing n.

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

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

Each H_j can be computed in $\tilde{O}(\mu(ut))$ using the standard product tree method described in Algorithm 1 (for $j = 0$) illustrated in Figure 5. 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 $\tilde{O}(\frac{n}{t}\mu(ut))$.

What is the caveat?

The same technique applies to factorization[6], but with a slight *caveat*.

After computing the tree product, we can compute the residues of a modulo H_0. Then we can compute the residues of $a \bmod H_0$ modulo the two children π_2 and π_3 of $H_0 = \pi_1$ in the product tree (depicted in Figure 5), and so on. Intuitively, we descend the product tree

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

Algorithm 1 Product Tree Algorithm

Require: the set h_i
Ensure: $\pi = \pi_1 = \prod_0^{t-1} h_i$, and π_i for $i \in \{1, \ldots, 2t-1\}$ as in Figure 5

```
π ← array of size t
function prodTree(i,start,end)
    if start = end then
        return 1
    else if start + 1 = end then
        return h_start
    else
        mid ← ⌊(start+end)/2⌋
        π_{2i} ←prodTree(2i,start,mid)
        π_{2i+1} ←prodTree(2i + 1,mid,end)
        return π_{2i} × π_{2i+1}
π_1 ←prodTree(1, 0, t)
```

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 ($i \in \{0, \ldots, t-1\}$). This is described in Algorithm 6 and illustrated in Figure 6. 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 ($i \in \{jt, \ldots, jt+t-1\}$) 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 $\tilde{O}(\frac{n}{t}\tilde{O}(\mu(ut)))$.

Does it make sense to nest $\tilde{O}$'s?

Algorithm 2 Division Using a Product Tree

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

```
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)
```

4.2 Adapting p

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 and all the h'_i. We can hence consider the following variants:

What is the goal of the following variants? Is it to reduce the complexity of the generation of the primes?

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

Variant 1: Smooth p_i:

$$p_i = \prod_{j=r_i}^{r_{i+1}-1} \texttt{Prime}[j]$$

Where the bounds r_i are chosen to ensure that each p_i has the proper size.

Variant 2: $p_i = \texttt{Prime}[i]^{r_i}$ where the exponents r_i are chosen to ensure that each p_i has the proper size.

Variant 3: Progressively work modulo a growing power of two. This variant, is probably the most efficient of all, but somewhat complex to explain. We hence describe it in detail in Appendix B. This variant is compatible with the use of FFT-multiplication, hence asymptotic complexity is preserved. In addition, it avoids all modular reductions and all CRT re-combinations and hence offers considerable constant-factor accelerations.

```
Have this written by Fabrice...

(les modulos etant $2^{ut}, 2^{2ut}, ...$):
- calcul des $H_j$ comme dans 4.1 par product tree
- produit des $H_j$ de la facon suivante pour round 1:
  - $\mbox{prod} = 1$ (de taille $ut$)
  - pour $j=1,...$
      $\mbox{carry}_j | \mbox{prod} = \mbox{prod} \times H_j$
  - retourner prod, et garder carry
- produit des $H_j$ pour round suivants:
  - prod = 0
  - pour $j=1,...$
      $\mbox{carry}_j | \mbox{prod} = \mbox{prod} \times H_j + \mbox{carry}_j$

L'idee est la suivante (invariant): au round i et au tour j, prod contient les bits $(i-1)ut,.
$\mbox{carry}_j$ contient le carry genere par ces bits...
```

5 Implementation

We implemented and benchmarked the Divide & Factor protocol described in the previous sections. The implementation is called `btrsync`, its source code is available from [6].

on parle de "in the previous sections" mais de quelle variate s'agit il?

The program performs unidirectional synchronization, which is simpler to understand. The code is divided into two subprograms: a shell script and a Python program:

The Shell Script sets up two instances of the Python program on Oscar and Neil and establishes a bidirectional communication channel between them using two Unix pipes between their standard inputs and outputs.

The Python Program uses `gmp` to perform all the number theory operations and performs the actual synchronization. It proceeds in two phases:

Finding Different Files

1. Hash of files' contents concatenated with their paths, types (folder/file), and permissions (not supported yet).
2. Implement the protocol proposed in Section **??** with input data coming from `stdin` and output data going to `stdout`.

encore une fois "Implement the protocol proposed in Section **??***" qu'a t on implemente au juste?*

sur "a first prime number p_1", l'implem marche avec p_i ou des 2^u?

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 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 files. 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...

5.1 Move Resolution Algorithm

To reproduce the structure of Oscar on Neil's disk, we need to perform a sequence of file moves. Sadly, moves are 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 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 3.

An optimization implemented by `btrsync` over the algorithm described here is to move files instead of copying them and then remove the original file. Moves are faster than copies on most filesystems as the OS does not need to copy the actual file contents to perform moves.

5.2 Experimental Comparison to `rsync`

We compared `rsync`[8] and our Divide & Factor implementation (called `btrsync`) under the following experimental conditions:

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

Algorithm 3 Perform Moves

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

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

Test Directories: The directories used for transmission and time comparisons are described in Table 3.

Command-Line Options: `rsync` was called with the following options, for the reasons below:

▶ `--delete` to delete existing files on Oscar which do not exist on Neil like `btrsync` does.

▶ `-I` to ensure that `rsync` did not cheat by looking at file modification times (which `btrsync` does not do).

▶ `--chmod="a=rx,u+w"` in an attempt to disable the transfer of file permissions (which `btrsync` does not transfer). Although these settings ensure that `rsync` does not need to transfer 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` Transmission accounting was performed by calling `rsync` with the `-v` flag (which reports 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.

Network Configuration: Experiments were performed without any network transfer, by synchronizing two folders on the same host. Hence, time measurements should mostly represent the CPU cost of the synchronization.

Results: Results are given in Table 2. 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 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.

6 Conclusion and Further Improvements

The main contributions of our work are:

Vérifier les claims de cette liste, et en parler dès l'intro.

- We present the novel "Divide & Factor" protocol for set reconciliation, which is based on number theory and is optimal with respect to transfer size.
- We study the problem of set reconciliation of directories of files. We discuss the optimal size of message digests in this setting, as well as a move resolution algorithm to reproduce a directory structure.
- We present `btrsync`, an open source implementation of the "Divide & Factor" protocol.
- We demonstrate the usability of this implementation through benchmarks on synthetic and real-world tasks, and show that `btrsync` exchanges less data than the popular software `rsync`.
- The optimizations presented in this paper apply to [] as well.

TODO résoudre ce point.

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.

7 Acknowledgment

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

8 ToDo

- Pourquoi on a deux biblios?! Laquelle est la bonne ?
- Merge two reference files rsynch and wagner.
- @Fabrice: Fig. 1 pas clair, je pense qu'il faut mieux créer une notation mu gcd car en fait la FFT ne modifie que le mu... Et on pourrait plutôt indiquer dans ce tableau la complexité (avec mu et mu gcd) en utilisant l'optimisation décrite ou sans utiliser l'optimisation décrite). N'oublions pas de préciser que les temps des expériences d'Antoine comptent aussi le tirage des nombres premiers (qui doit être négligeable peut-être dans notre cas, je ne me rappelle plus...)
- @Fabrice: 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...
- @Fabrice: il faut discuter de la taille de la taille maximale des "petits" premiers utilisés pour les variantes de p et montrer que cela n'enlève pas trop d'entropie pour les h_i. Encore une fois, je m'en occupe la semaine prochaine si besoin.
- Refaire une dernière fois les expériences, vu que Fabrice a significativement amélioré les perfs.
- Faire clarifier par Fabrice l'histoire du doublement...
- Prendre en compte les remq suivants de Fabrice

```
- je pense que l'événement bottom_bandwidth est ou bien l'événement "on n'a pas réussi à fa

- je pense que tu essayes de traiter 2 cas ensembles: modulus trop petit (t0 > t) et collis
- dans l'intro, au lieu de "This article proposes a new reconciliation protocol based on nu
of asymptotic transmission complexity, the proposed procedure reaches optimality as well."
j'aurai dis quelque chose comme:
"In this article, we first propose a new reconciliation protocol based on number theory. In
of asymptotic transmission complexity, the proposed procedure reaches optimality as well.

Then, we show how to apply this set reconciliation protocol to efficiently synchronize real
We show that the naive method of hashing the files and synchronizing these hashes can be im
First, we show that the size of the hashes can be carefully optimize to reduce the bandwidt
Then, we show that a protocol to detect moves of files and carefully use this information t
Finally, we run some practical experiments and compare ... rsync ..."
```

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, *Noisy Polynomial Interpolation and Noisy Chinese Remaindering*, Advances in Cryptology – Proceedings of Eurocrypt 2000, 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/Btrsync`

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 $c \bmod p_1p_2 = \text{CRT}_{p_1,p_2}(c_1, c_2)$
		computes a, b using p_1p_2
		if a factors properly then go to Final Phase
		else perform the protocol with p_3
	$\xrightarrow{c_3}$	
		computes $c \bmod p_1p_2p_3 = \text{CRT}_{p_1,p_2,p_3}(c_1, c_2, c_3)$
		computes a, b using $p_1p_2p_3$
		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_is 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 Power of Two Protocol

In this variant Oscar computes c in $\mathbb{N}$:

$$c = \prod_{F_i \in \mathfrak{F}} \texttt{HashPrime}(F_i) = \prod_{i=1}^{n} h_i \in \mathbb{N}$$

and considers $c = \bar{c}_{n-1}|\dots|\bar{c}_2|\bar{c}_0$ as the concatenation of n successive u-bit strings. Again, we omit the treatment of $\perp$s for the sake of clarity.

First phase during which Neil amasses modular information on the difference		
Oscar		**Neil**
computes $c \in \mathbb{N}$		
	$\xrightarrow{\bar{c}_0}$	
		computes a, b modulo 2^u if a factors properly then go to Final Phase else request next chunk $\bar{c}_1$
	$\xrightarrow{\bar{c}_1}$	
		construct $c \bmod 2^{2u} = \bar{c}_1\|\bar{c}_0$ computes a, b modulo 2^{2u} if a factors properly then go to Final Phase else request next chunk $\bar{c}_2$
	$\xrightarrow{\bar{c}_2}$	
		construct $c \bmod 2^{3u} = \bar{c}_2\|\bar{c}_1\|\bar{c}_0$ computes a, b modulo 2^{3u} if a factors properly then go to Final Phase else request next chunk $\bar{c}_3$
	$\vdots$ (for $2t$ rounds)	

Final Phase		
		Let $\mathfrak{S} = \{F'_i \text{ s.t. } a \bmod 2^{2tu} = 0\}$
	$\xleftarrow{\mathfrak{S},b}$	
deletes files s.t. $b \bmod 2^{2tu} = 0$ adds $\mathfrak{S}$ to the disk		

C Hashing Into Primes

Hashing into primes is frequently needed in cryptography. A recommended implementation of `HashPrime`(F) is given in Algorithm 5. If u is large enough (*e.g.* 160) one might sacrifice uniformity to avoid repeated file hashings by defining `HashPrime`(F) = `NextPrime`(`Hash`(F)). Yet another acceleration (that further destroys uniformity) consists in replacing `NextPrime` by Algorithm 4 where $\alpha = 2 \times 3 \times 5 \times \cdots \times \texttt{Prime}[d]$ is the product of the first primes until some rank d.

Antoine sur quelle plate-formes ont ete obtenus les resultats experimentaux? quelles vitesses de processeur etc?

Algorithm 4 Fast Nonuniform Hashing Into Primes

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

Algorithm 5 Possible Implementation of $\mathtt{HashPrime}(F)$

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

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`	1613	778353	1846	788357	10237	1.0	0.2	7.7
`empty`	`source`	11	29	12436	6120	18516	463.9	0.1	5.5
`empty`	`empty`	11	29	19	28	7	1.2	0.1	0.3
`synthetic`	`synthetic_shuffled`	24891	51019	3638	4147	-68125	0.1	0.2	26.8
`synthetic_shuffled`	`synthetic`	24701	50625	3443	3477	-68406	0.9	0.2	26.6
`synthetic`	`synthetic`	25011	50918	327	28	-75574	0.0	0.1	25.7
`firefox-13.0.1`	`firefox-13.0`	90598	28003573	80895	27995969	-17307	1.0	2.6	4.2
`source_moved`	`source`	2456	694003	1603	1974	-692882	0.0	0.2	2.5

Table 2. 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.

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 3. Test Directories.

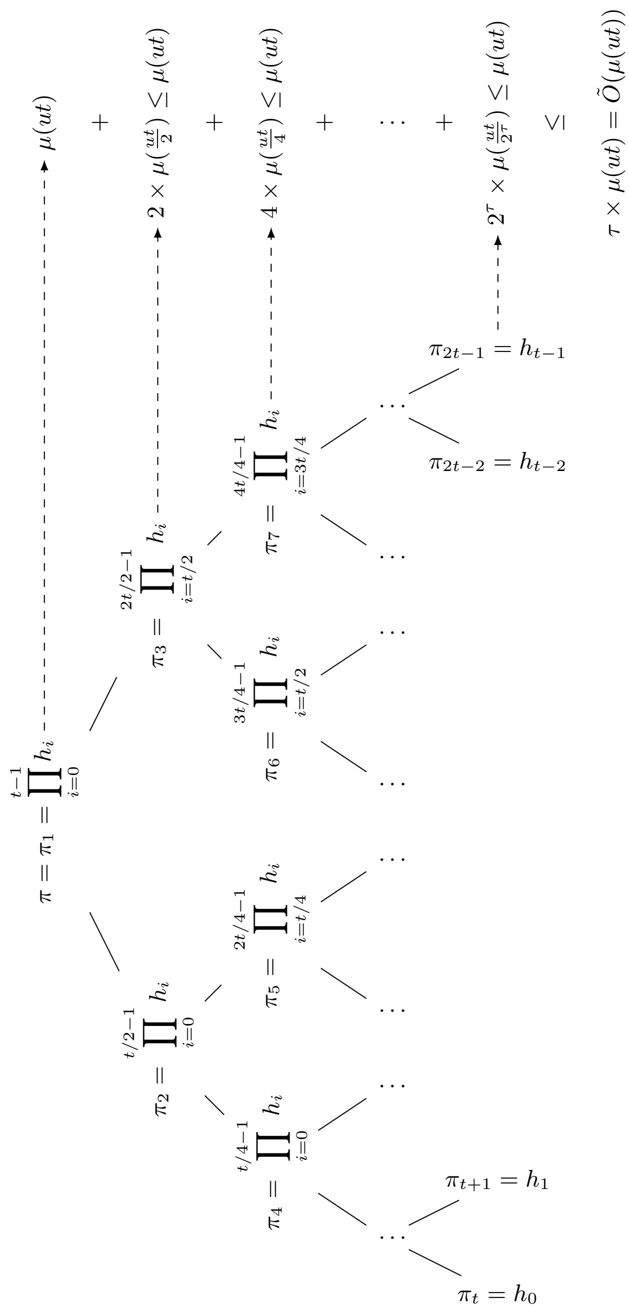

Fig. 5. Product Tree

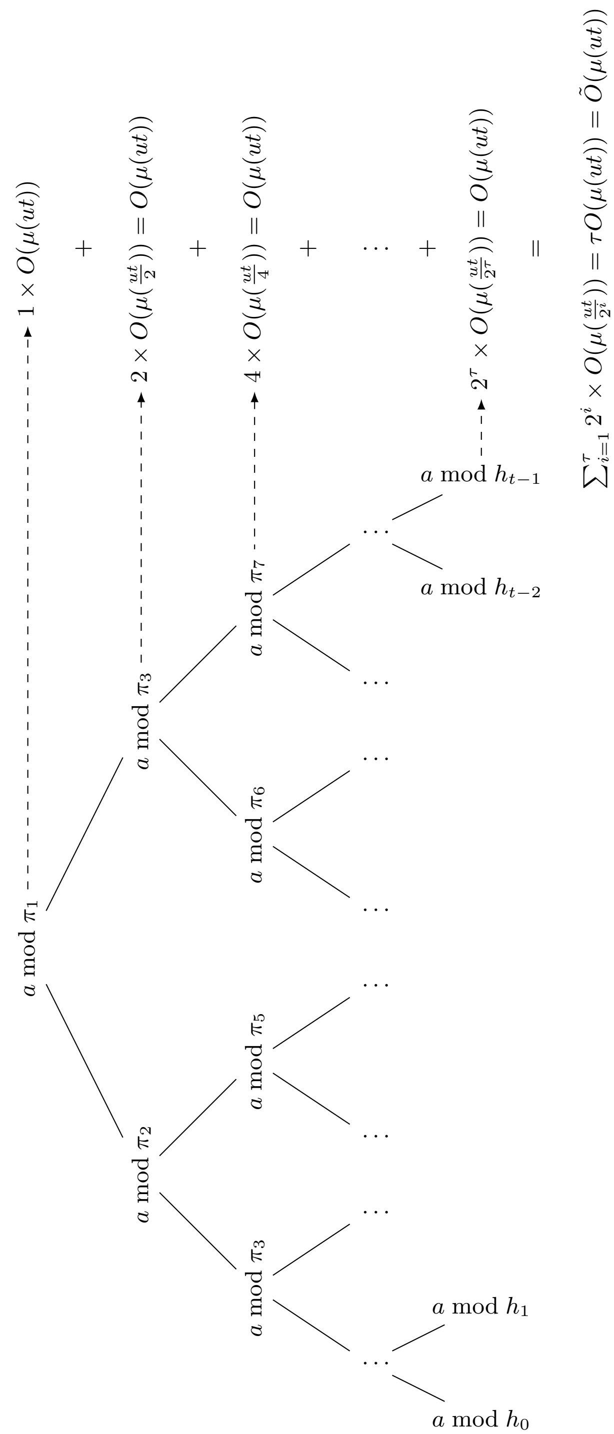

Fig. 6. Modular Reduction From Product Tree