When File Synchronization Meets Number Theory

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

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

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

1 Introduction

In this work we [to be completed by David]

2 A Few Notations

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

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

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

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

3 The Content Synchronization Protocol

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

TODO(amarilli) we need a real analysis of the expected quantity of information transferred. Since apparently there is a hope that we are better than them (because we can adapt the size of the hashes), we should really make this clearer than what the previous paragraph says, ie. reformulate the problem as a synchronization problem where hash sizes are unknown so that we are better than them.

---

[1] Oscar and Neil will respectively stand for *o*ld and *n*ew.

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

3.1 Description of the Basic Exchanges

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

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

We generate a prime p such that:

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

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

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

Oscar computes:

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

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

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

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

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

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

Formally, this is done as follows:

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

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

3.2 The Case of Insufficient Information

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

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

4 Efficiency Considerations

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

4.1 Probabilistic Decoding: Reducing p

Generate a prime p smaller than previously, namely:

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

for some small integer $w \geq 1$ (say $w = 50$). For large $\eta = \max(n, n')$ and t, the size of the new prime p will be approximately half the size of the prime p generated in section 3.1. The resulting redundancy c is calculated as previously but is approximately twice smaller. As previously, we have:

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

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

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

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

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

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

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

To make the heuristic synchronization deterministic, the parties can can use the $\texttt{Hash}(\mathfrak{F}')$ protocol preamble mentioned in section 3.2.

4.2 The File Laundry: Reducing u

What happens if we shorten u in the basic protocol?

TODO(amarilli): I find it hard to understand what we are studying, and why we are interested in the probability of a file to collide in all rounds. I get it now, but maybe we can improve the writing.

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

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

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

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

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

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

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

By independence, we have

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

Therefore the number of files which collide is, on average:

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

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

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

For the $(\eta = 10^6, u = 32)$ instance considered previously we get $\epsilon_2 \leq 5.43\%$ and $\epsilon_3 \leq 2 \cdot 10^{-3}\%$.

Improvement As mentioned previously, the parties can remove the files revealed as different during the first possible iteration and only work with common and colliding files. Now, the only collision which can be bad for round k, are the collisions of a file i with a file j such that i and j both have collided at all the previous iterations. And let write Z_i^ℓ the random variable equal to 1 when the file i has a bad collisions for all the ℓ first iterations.

Suppose $\eta > 1$. Let us set $Z_i^0 = 1$ and let us 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

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

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

Furthermore $p_0 = 1$ and

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

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 probability that after k rounds at least one false positive will survive is

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

For the $(\eta = 10^6, u = 32, k = 2)$ instance considered previously we get $\epsilon_2 \leq 0.013\%$.

TODO: verify I have not made a mistake and compare with using a bigger u (maybe using example... and timing...)

5 Theoretical time complexity and algorithmic improvements

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

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

5.1 Theoretical complexity

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

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

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

5.2 Improvements

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

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

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

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

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

Algorithm 1 Product tree algorithm

Require: a table h such that $h[i] = h_i$
Ensure: $\pi = \pi_1 = \prod_0^{t-1} h_i$, and $\pi[i] = \pi_i$ for $i \in \{1, \ldots, 2t-1\}$ as in Figure **??**

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

Algorithm 2 Division using product tree

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

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

6 Optimizing Parameters

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

6.1 Using a Smooth p

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

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

7 Implementation

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

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

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

7.1 The Bash Script

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

7.2 The Python Script

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

Finding Different Files

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

More precisely:

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

[please structure the following:]

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

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

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

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

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

Capture the following:

integration de tout le code d'Antoine qui permet de deplacer les fichiers du cote de Neil, et evite des synchronisations inutiles de fichiers deja presents du cote de Neil. L'algorithme est plus complexe qu'il n'en a l'air car il faut gerer les cycles de deplacements, ("a" renomme en "b" lui-meme renomme en "a")... On notera que le code Haskell ne prenait pas en compte les cycles notamment (et donc etait buggue). D'une maniere tres amusante, on peut voir l'algo propose par Antoine comme une decomposition en composantes fortement connexes du graphe des deplacements (si on oublie le cas des repertoires – les noeuds sont les fichiers, les aretes sont les deplacements a effectuer). Et l'algorithme consiste a effectuer d'abord les deplacements des composantes filles avant ceux des composantes parentes... On remarquera que comme chaque noeud a au plus un antecedent, on peut montrer tres facilement que les composantes fortement connexes sont des noeuds seuls ou des cycles. Pour traiter les cycles, l'algorithme stocke un des fichier dans un repertoire temporaire. utilisation du json pour l'envoi des messages entre Neil et Oscar (beaucoup plus sur que les eval que je faisais – cependant, cela fait perdre quelques octets vu la maniere dont j'ai code...) TODO: benchmarks meilleure integration du code d'Antoine pour eviter le recalcul des hashes gestion des fichiers identiques du cote de Neil (l'algo des deplacements ne gere que le cote d'Oscar): si deux fichiers sont identiques du cote de Neil, il est inutile de transferer les deux. C'est trivial a implementer, mais il est tard

7.3 Program Structure

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

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

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

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

7.4 Time Measurements

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

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

8 Conclusion and Further Improvements

In this work we [to be completed by David]

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

9 Acknowledgment

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

todo: Fix euclidean to Euclidean in reference 5.

todo: Merge two reference files rsynch and wagner.

References

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

A Extended Protocol

First phase during which Neil amasses modular information on the difference

Oscar		**Neil**
		start protocol with p_1
	$\xrightarrow{c_1}$	
		computes a, b using p_1
		if a factors properly then Final Phase
		else perform protocol with p_2
	$\xrightarrow{c_2}$	
		computes $c \bmod p_1p_2 = \mathrm{CRT}_{p_1,p_2}(c_1, c_2)$
		computes a, b using p_1p_2
		if a factors properly then Final Phase
		else perform protocol with p_3
	$\xrightarrow{c_3}$	
		computes $c \bmod p_1p_2p_3 = \mathrm{CRT}_{p_1,p_2,p_3}(c_1, c_2, c_3)$
		computes a, b using $p_1p_2p_3$
		if a factors properly then Final Phase
		else perform protocol with p_4
	$\vdots$	

Final Phase

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

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