When File Synchronization Meets Number Theory

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

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

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. Note that in reality $\mathfrak{D}$ and $\mathfrak{D}'$ may differ both in their files and in their tree structure.

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

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

3 The Contents 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 and novelty.

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:

[1] Oscar and Neil will respectively stand for *o̲ld* and *n̲ew*.
[2] *e.g.* SHA-1

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

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

$$c = \prod_{i=1}^{n} h_i \bmod p \tag{2}$$

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 :

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

Note that since $\mathfrak{F}$ and $\mathfrak{F}'$ differ by at most t elements, we have that a and b are strictly lesser than 2^{ut}. Theorem 1 (see [2]) shows that given s one can recover a and b efficiently. The algorithm is based on Gauss' algorithm for finding the shortest vector in a two-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 be some prime integer such that $2AB < p$. Let $s = ab^{-1} \mod p$. Then given A, B, s and p, one can recover a and b in polynomial time.*

@fbenhamo: TODO REMARK: Actually, this problem is known as rational number reconstruction [4] and [8].

Taking $A = B = 2^{ut} - 1$, we have from (1) that $2AB < p$. Moreover, $0 \leq a \leq A$ and $0 < b \leq B$. Therefore, we 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 $\mathfrak{F}'$.

Formally, this is done as follows:

Oscar		**Neil**
	$\xrightarrow{c}$	
		computes a, b
		if a does not factor properly
		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 if bounds on parameter sizes are respected. However, a file synchronization procedure that works *only* for a limited number of differences is not of major practical usefulness. In the next subsection we will show how to extend the protocol even in the case where the differences exceed the informational capacity of the modulus p used.

3.2 The Case of Insufficient Information

To extend the protocol to an unlimited number of differences, Oscar and Neil will use more than one p by agreeing on an infinite set of primes $p_1, p_2, \ldots$ As long as the protocol fails, Neil will keep amassing information about the difference as shown in the appendix. Note that no information is lost and that information adds-up until it reaches a threshold that suffices to identify the difference.

4 Variants

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

4.1 Probabilistic Decoding: Reducing p

Generate a prime p smaller than previously, namely:

$$2^{ut+w-1} < p \leq 2^{ut+w} \tag{3}$$

for some small integer $w \geq 1$ (we suggest to take $w = 50$). 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 the previous section. The resulting redundancy c is calculated as previously but approximately twice smaller. As previously, we have :

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

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

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

The difference with respect to the basic protocol is that we don't have a fixed bound for a and b anymore; equation (4) only provides a bound for the product ab. Therefore, we define a finite sequence of integers:

$$(A_i = 2^{w \cdot i}, B_i = \lfloor (p-1)/(2 \cdot A_i) \rfloor) \quad \text{where} \quad B_i > 1$$

.

$\forall i > 0$ we have $2A_iB_i < p$. Moreover, From equations (3) and (4) there must be at least one index i such that $0 \le a \le A_i$ and $0 < b \le B_i$. Then using Theorem 1, given (A_i, B_i, p, s) one can recover a and b, and eventually the difference.

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 (4) 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.

4.2 File Laundry: Reducing u

What happens if we shorten u in the basic protocol?

First method As foreseen by the birthday paradox, we should start seeing collisions. Let us analyse them.

We see the hash function `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 number i has a collision with another file, and equal to 0 otherwise. Clearly, we have $\Pr\left[\, X_i = 1 \,\right] \le \frac{\eta-1}{2^u}$. And the number of files which collides are, in average:

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

For instance, for $\eta = 10^6$ files and 32-bit hash values, one should expect about at most 233 files which collide.

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

Hence, it suffices to replace the function $\mathtt{Hash}(F)$ by a chopped $\mathtt{MAC}_k(F) \bmod 2^u$ 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 k of iterations, if the hash functions are random and independent for each iteration. Assume the η remains invariant between iterations. 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. 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. By independence, we have

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

Therefore the number of files which collides is, in average

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

---

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

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, this gives $\epsilon_2 \leq 5.43\%$ and $\epsilon_3 \leq 2 \cdot 10^{-3}\%$.

Improvement However, we can improve a lot the algorithm, using the following trick: we can remove the files which are different in the first possible iteration, and only work with common files and files which collided (in a bad way, blinding a difference). 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 $p_l = \Pr\left[Z_i^{l-1} = 1 \text{ and } Z_j^{l-1} = 1 \right]$ for all l and $i \neq j$. For $k \geq 1$, we have

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

Furthermore $p_0 = 1$ and

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

so we have

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

and

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

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 ($\eta = 10^6, u = 32$) instance considered previously, this gives $\epsilon_2 \leq 0.013\%$.

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

5 Theoretical complexity and algorithmic improvements

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

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 about $\tilde{O}(M(k))$ [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 $10^4 * 64$ 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 and, 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

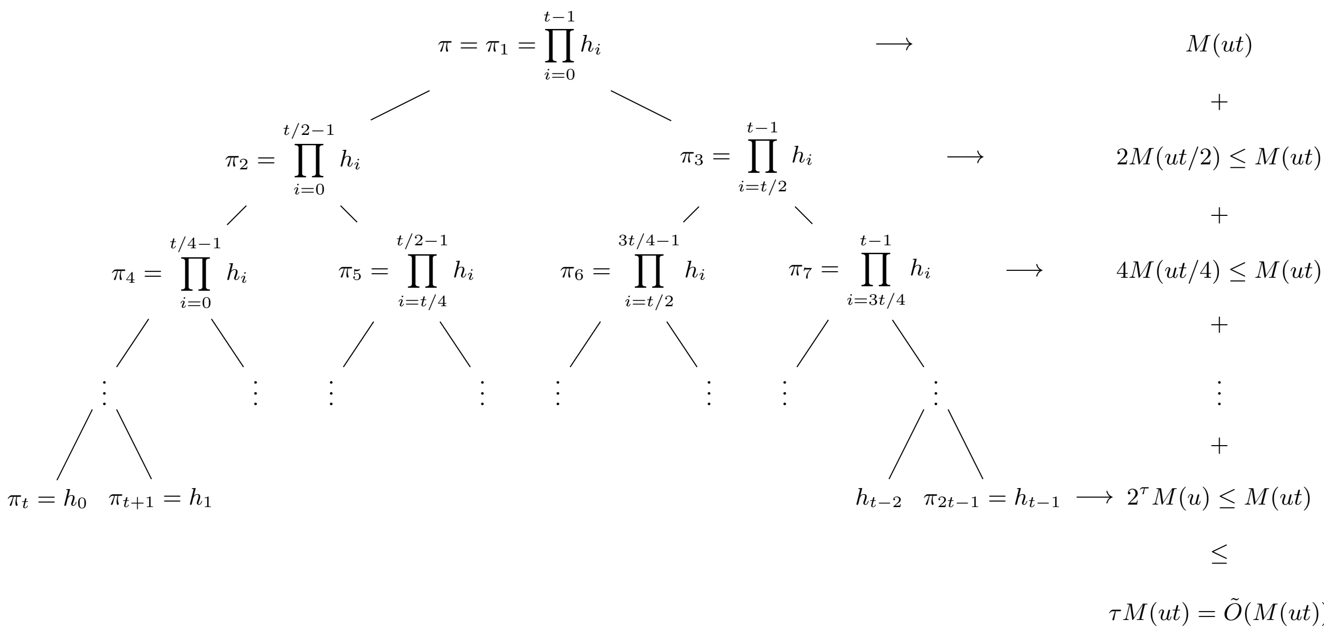

Fig. 1. Product tree

5.2 Improvement

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

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

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

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

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

Algorithm 1 Product tree algorithm

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

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

Algorithm 2 Division using product tree

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

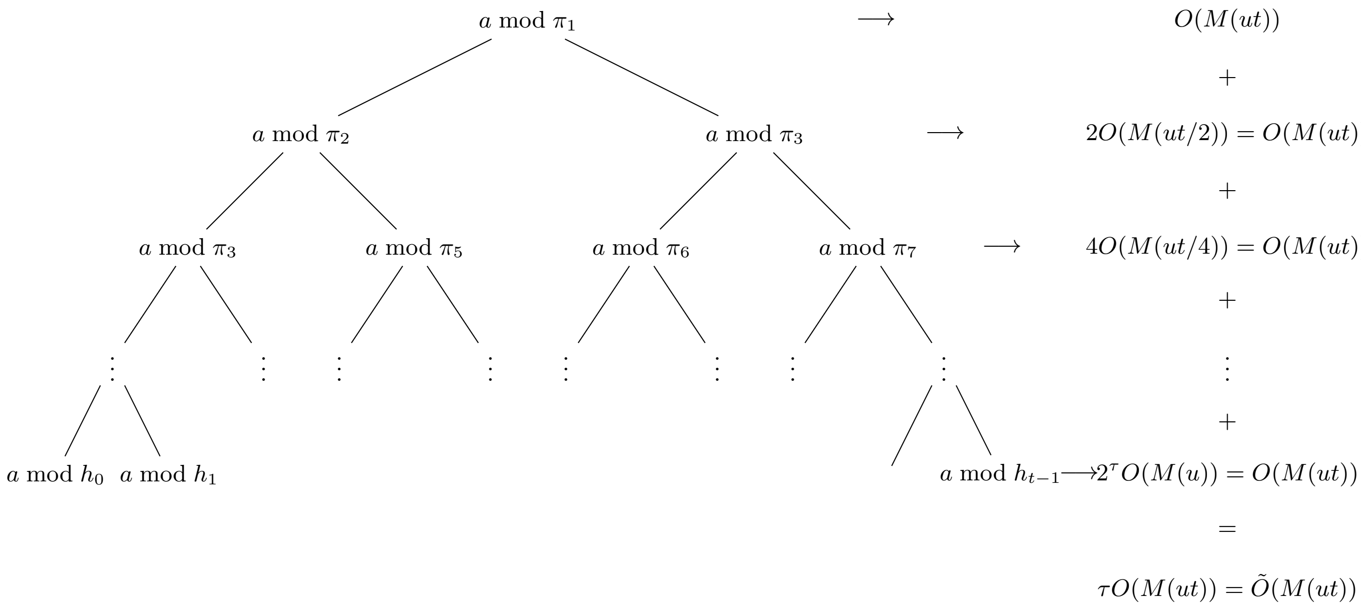

Fig. 2. Division from product tree

6 Implementation

6.1 Program Structure

6.2 Time Measurements

7 Conclusion and Further Improvements

In this work we [to be completed by David]

8 Acknowledgment

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

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 Terminate Phase
		else switch to 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 Terminate Phase
		else switch to 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 Terminate Phase
		else switch to p_4 ...
	$\vdots$	

Terminate 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 bits of each p_i could be generated using a pseudo-random number generator and a small corrected additive constant of an average value of $\ln(p_i) \cong \ln(2^{2tu+2}) \cong 1.39(tu+1)$ which storage requires essentially $\log_2(tu)$ bits.